S.Ha.R.K. User Manual

Volume IV

S.Ha.R.K. Kernel Architecture

Written by

Giorgio Buttazzo (giorgio@sssup.it)

Paolo Gai (pj@sssup.it)

Luigi Palopoli (luigi@hartik.sssup.it)

Marco Caccamo (caccamo@sssup.it)

Giuseppe Lipari (lipari@sssup.it)

Scuola Superiore di Studi e Perfezionamento S. Anna

RETIS Lab

Via Carducci, 40 - 56100 Pisa

Contents

1 Kernel Overview **6**
- 1.1 Scheduling Architecture . . . 6
- 1.2 QoS Specification . . . 8
- 1.3 Scheduling Modules . . . 10
  - 1.3.1 Module Organization . . . 10
  - 1.3.2 Sample scheduling architectures . . . 11
  - 1.3.3 Module Interface . . . 13
  - 1.3.4 Public Functions . . . 13
  - 1.3.5 Private Functions . . . 13
  - 1.3.6 An Example . . . 13
- 1.4 Resource Modules . . . 14
- 1.5 Shared Resource Access Protocols . . . 14
- 1.6 The Generic Kernel . . . 17
  - 1.6.1 Task descriptor . . . 17
  - 1.6.2 Task States . . . 18
- 1.7 Initialization and termination of the system . . . 18

2 Models **20**
- 2.1 Data structures . . . 20
- 2.2 Task Models . . . 20
  - 2.2.1 `TASK_MODEL` . . . 21
  - 2.2.2 Used conventions . . . 24
  - 2.2.3 Examples of Models currently integrated into the Kernel . . . 24
- 2.3 Resource Models . . . 26
- 2.4 Mutex attributes . . . 26

3 Kernel Internals **27**
- 3.1 Kernel types . . . 27
- 3.2 Descriptors . . . 27
  - 3.2.1 Task Descriptor . . . 28
  - 3.2.2 Level descriptor . . . 29
  - 3.2.3 Resource module descriptor . . . 30
- 3.3 System states . . . 30
- 3.4 Kernel Global Variables . . . 31
- 3.5 Temporal protection . . . 32
  - 3.5.1 Negative capacities . . . 33
- 3.6 Utility functions . . . 33

3.6.1 Event Handling . . . 33
3.6.2 Exception Handling . . . 34
3.6.3 Memory Management . . . 34
3.6.4 Context switch . . . 35
3.6.5 Queues, arrays and pointers . . . 35
3.6.6 Initialization functions . . . 37
3.6.7 Interrupt disabling, printf and system termination . . . 38

4 The Scheduling Modules **39**
4.1 Task Lifecycle . . . 39
4.2 Assumptions on Task Queues . . . 41
4.3 Scheduling Module Interface . . . 41
4.3.1 Public Functions . . . 42
4.3.2 Private Functions . . . 47
4.4 Registration Function . . . 48
4.4.1 Default values . . . 49
4.5 Writing Conventions . . . 50

5 Resource Modules **51**
5.1 Resource Modules Interface . . . 51
5.2 Implementation of the Shared Resource Access Protocols . . . 52
5.2.1 Used Approach . . . 52
5.2.2 The mutexes . . . 52
5.2.3 Interface extension . . . 54

6 Examples **55**
6.1 SIMPLEEDF (Earliest Deadline First) Scheduling Module . . . 55
6.1.1 State transition diagram . . . 55
6.1.2 Level descriptor . . . 56
6.1.3 Task descriptor . . . 57
6.1.4 Module internal event handlers . . . 58
6.1.5 Public Functions . . . 58
6.1.6 Private Functions . . . 60
6.1.7 Additional Functions exported by the Module . . . 61
6.2 PS (Polling Server) Scheduling Module . . . 61
6.2.1 Transition state diagram . . . 62
6.2.2 Private Data structures . . . 62
6.2.3 Internal Module Functions . . . 63
6.2.4 Public Functions . . . 64
6.2.5 Task Calls . . . 64
6.2.6 Private functions . . . 66
6.2.7 Functions exported by the Module . . . 66
6.3 PI (Priority Inheritance) Resource Module . . . 66
6.3.1 Used approach . . . 66
6.3.2 Private data structures . . . 67
6.3.3 Internal and Interface Functions . . . 68

7 The Generic Kernel Internals **70**
7.1 System Tasks and User Tasks . . . 70
7.2 Initialization and Termination . . . 71
7.2.1 Interrupt Disabling . . . 71
7.2.2 Initialization of the Memory Management . . . 71
7.2.3 Initialization of the static data structures . . . 72
7.2.4 Resgistration of the Modules in the system . . . 72
7.2.5 OS Lib initialization . . . 73
7.2.6 Initialization functions call . . . 73
7.2.7 First context change . . . 73
7.2.8 The shutdown functions . . . 74
7.2.9 Termination request for all user tasks . . . 74
7.2.10 Second context change . . . 74
7.2.11 Exit functions called before OS Lib termination . . . 74
7.2.12 Termination of the OS Lib . . . 74
7.2.13 Exit functions called after OS Lib termination . . . 75
7.3 Task creation and on-line guarantee . . . 75
7.4 Task activation . . . 76
7.5 The Scheduler . . . 77
7.5.1 Current slice end for the running task . . . 77
7.5.2 Scheduling . . . 78
7.5.3 Dispatching . . . 78
7.6 Execution Time statistics . . . 78
7.7 Cancellation . . . 80
7.7.1 The task_makefree function . . . 80
7.7.2 Cancellation point registration . . . 81
7.7.3 Cleanups and Thread Specific Data . . . 81
7.8 Signals . . . 81
7.9 Task Join . . . 82
7.10 Pause and Nanosleep . . . 83
7.11 Mutex and condition variables . . . 83
7.12 Other primitives . . . 84

Chapter 1

Kernel Overview

1.1 Scheduling Architecture

S.Ha.R.K. is a research kernel expressly designed to help the implementation and testing of new scheduling algorithms, both for the CPU and for shared resources. In order to achieve independence between applications and scheduling algorithms (and between the schedulers and the kernel), S.Ha.R.K. is based on a *Generic Kernel*, which does not implement any particular scheduling algorithm, but postpones scheduling decisions to external entities, the *scheduling modules*. In a similar fashion, the access to shared resources is coordinated by *resource modules*. External modules can implement periodic scheduling algorithms, soft task management through real-time servers, semaphore protocols, and resource management policies. The modules implementing the most common algorithms (such as RM, EDF, Round Robin, and so on) are already provided, and it is easy to develop new modules. A scheme of the kernel architecture is depicted in Figure 1.1.

The OS Lib layer implements an abstraction of a generic machine capable of exporting some services, as for example context switching, time management, interrupts handling, and a subset of the run-time C library. For detailed information about the OS Lib, see [1][2].

The Generic Kernel provides the mechanisms used by the modules to perform scheduling and resource management thus allowing the system to abstract from the specific algorithms that can be implemented. The Generic Kernel simply provides the primitives without specifying any algorithm, whose implementation resides in external modules, configured at run-time with the support of the *Model Mapper* (see Section 1.2).

Another important component of the Generic Kernel is the Job Execution Time (JET) estimator, which monitors the computation time actually consumed by each job. This is a generic mechanism, independent from the scheduling algorithms, that can be used for statistical measurements, for enforcing temporal protection, or for resource accounting (see Section 7.6).

The API is exported through the *Libraries*, which use the Generic Kernel to support some common hardware devices (i.e., keyboard, sound cards, network cards, graphic cards) and provide a compatibility layer with the POSIX Real-time Controller System Profile [10].

An Application can be considered as a set of cooperating tasks[1] that share a common address space. There is no memory protection implemented into the Kernel. Intertask communication is performed using shared memory buffers accessed through some synchronization mechanisms (as

[1]In this document, a process is a set of computations performed in a private address space. A thread is a single execution flow in a process. Each thread is identified by an unique ID, plus some specific parameters. The term Task is used as a synonymous of the term thread.

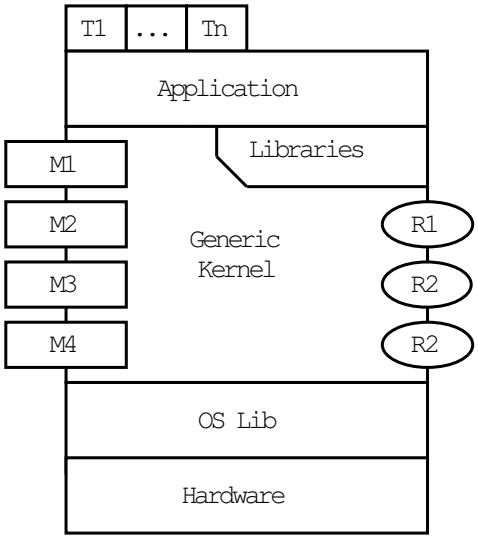

Figure 1.1: The S.Ha.R.K. Architecture

mutexes, condition variables, semaphores, CAB, message queues). Each task is characterized by a Task Model, some optional Resource Models, and a body. The body of a task is simply a C function with the prototype `void *body(void *arg)`.

An Application can use the following sets of functions:

- The functions exported by the OS Lib;
- The Generic Kernel primitives;
- Some Module-dependent functions;
- Some functions exported by libraries, device drivers, or the standard C library;
- The library implementing of the POSIX standard interface.

Each Module consists of a set of data and functions used for implementing a specific algorithm, whose implementation is independent from the other modules in the system, thus realizing a trade-off between user-level and in-kernel schedulers. In this way, many different module configurations are possible. For example, a Polling Server can either work with a RM or an EDF scheduling Module without any modification.

Currently, S.Ha.R.K. provides two basic modules:

- modules that implement scheduling algorithms and aperiodic service policies (Scheduling Modules);
- modules that manage shared (hardware or software) resources (Resource Modules);

All resource access protocols, such as Priority Inheritance, are implemented as a mutex module whose interface is derived from the resource module interface. A POSIX mutex interface is also provided on top of the implemented protocols.

Each type of Module provides a well defined interface to communicate with the Generic Kernel (user programs do not directly interact with the modules). The interface functions are called by the Generic Kernel to implement the kernel primitives. When modules need to interact with the hardware (for example, the timer), they can use the service calls provided by the Generic Kernel.

Finally, each Module has some unique identifiers to allow the implementation of some consistency checks (for example, a module that implements a Total Bandwidth Server cannot work with Rate Monotonic).

1.2 QoS Specification

One of the goals of S.Ha.R.K. is to allow the user to easily implement and test novel scheduling algorithms. In particular, the kernel has been designed with the following objectives:

- achieve independence between the kernel mechanisms and the scheduling policies for tasks and resource management;
- configure the system at run-time by specifying the algorithms to be used for task scheduling and resource access;
- achieve independence between applications and scheduling algorithms.

These requirements are particularly useful for comparing the performance of similar algorithms on the same application. In fact, module independence allows the user to configure and test applications without recompile them, only relinking them.

Independence between applications and scheduling algorithms is achieved by introducing the concept of *model*. Each task asks the system to be scheduled according to a given Quality of Service (QoS) specified by a model. In other words, a model is the entity used by S.Ha.R.K. to separate the scheduling parameters from the QoS parameters required by each task. In this way, the kernel provides a common interface to isolate the task QoS requirements from the real scheduler implementation.

Models are descriptions of the scheduling requirements expressed by tasks. S.Ha.R.K. provides three different kinds of models:

Task Models. A task model expresses the QoS requirements of a task for the CPU scheduling. Requirements are specified through a set of parameters at task creation time. Some of the task requirements are mandatory (e.g., the stack size of a task), while others depend on the specific task model (e.g., a deadline). For this reason, all task models are characterized by a general common part, which can be extended by a model-dependent part. Usually, the model-dependent part abstracts from a specific scheduling algorithm (for instance, the concept of period or deadline is independent from a specific algorithm like EDF or RM). The task models have a function similar to the `pthread_attr_t` structure defined in the POSIX standard.

Resource Models. A resource model is used to define the QoS parameters relative to a set of shared resources used by a task. For example, the resource model can be used to specify the semaphore protocol to be used for protecting critical sections (e.g., Priority Inheritance, Priority Ceiling, or SRP). In other cases, the resource model can be used to specify a hardware resource scheduling algorithm (e.g. a File System Scheduling Algorithm).

Mutex Models. When a mutex semaphore is created, these Models are used to specify the resource access protocol to be used, in a way similar to that done with Task Models. The

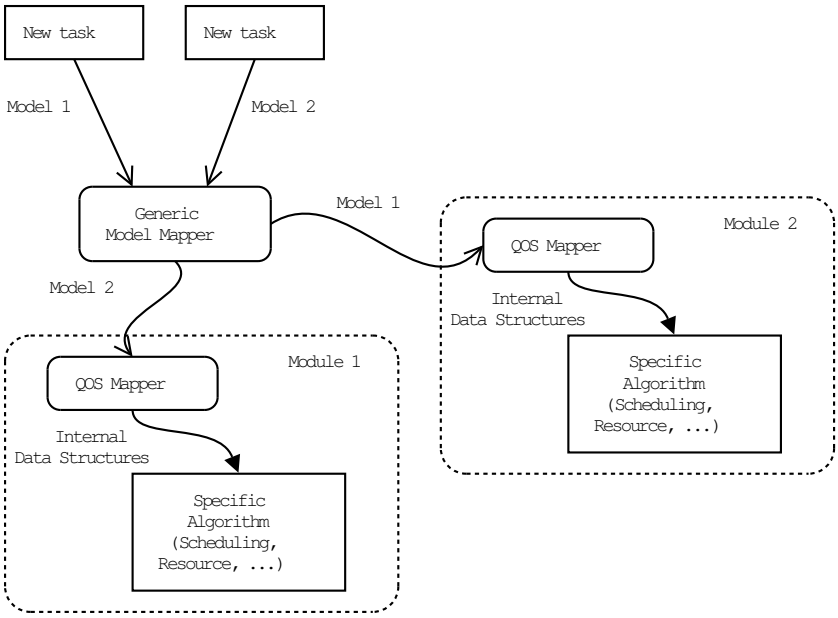

Figure 1.2: The interaction between the Model Mapper and the QOS Mapper.

mutex models have a function similar to the `pthread_mutexattr_t` structure defined in the POSIX standard.

Each task is characterized by a single mandatory QoS parameter, the *task criticality* (hard, soft, firm, non real-time, and so on). This parameter belongs the common part of the task model, together with a *model identifier* and some other parameters, such as the stack size.

Each task model is implemented as a C structure, in which the first field is the model identifier, the next fields are the mandatory parameters and the last field is a sequence of bytes containing the model-dependent parameters, that only the specific module can interpret. Resource models are completely generic, and depend on the resource they describe: the only mandatory parameter is the model identifier.

Models are required to make the generic kernel independent from the implemented scheduling algorithms: since the generic kernel does not implement any algorithm, it does not know how to serve a task but invokes a service request to scheduling entities realized as external *modules*. Hence, the generic kernel does not interpret the models, but just passes them to the modules; each module, by reading the common part of the model, can verify whether the task can be served or not.

Using models an application is able to specify the requested QoS, independently from the modules used into the system. For example, an application that creates a task using an Hard Task Model can be executed on an EDF, a RM, or a Deadline Monotonic Module.

Task creation works as follows (see Figure 1.2): when an application issues a request to the kernel for creating a new task, it also sends the model describing the requested QoS. A kernel component, namely the *model mapper*, passes the model to a module, selected according to an internal policy, and the module checks whether it can provide the requested QoS; if the selected module cannot serve the task, the model mapper selects a different module. When a module accepts to manage the task described by the specified model, it converts the model's

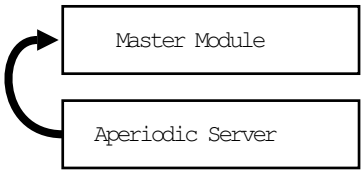

Figure 1.3: An aperiodic Server that inserts its tasks into a master module.

QOS parameters into the appropriate scheduling parameters. Such a conversion is performed by a module component, called the *QoS Mapper*. For example, a hard periodic task may have a model consisting of a period and a worst-case execution time (WCET); when a task is created with that model, the EDF module will convert such parameters into deadlines, reactivation times, and so on. In general, a module can manage only a subset of the models, and the set of models is not limited by the kernel. This is possible because the kernel does not handle the models, but it simply passes them to the Model Mapper, that selects a module and passes the model to that module. Currently, the Model Mapper uses a simple strategy, according to which modules are selected based on fixed priorities (see Section 1.3.1 for more details).

1.3 Scheduling Modules

Scheduling Modules are used by the Generic Kernel to schedule tasks, or serve aperiodic requests using an aperiodic server. In general, the implementation of a scheduling algorithm should possibly be independent on resource access protocols, and handle only the scheduling behavior. Nevertheless, the implementation of an aperiodic server relies on the presence of another scheduling module, called the Master Module (for example, a Deferrable Server can be used if the base scheduling algorithm is RM or EDF, but not Round Robin; see Figure 1.3). Such a design choice reflects the traditional approach followed in the literature, where most aperiodic servers insert their tasks directly into the scheduling queues of the base scheduling algorithm. Again, our modular approach masks this mechanism with the task models: an aperiodic server must use a task model to insert his tasks into the Master Module.

The Model Mapper distributes the tasks to the registered modules according to the task models the set of modules can handle. For this reason, the task descriptor includes an additional field (`task_level`), which points to the module that is handling the task.

When the Generic Kernel has to perform a scheduling decision, it asks the modules for the task to schedule, according to fixed priorities: first, it invokes a scheduling decision to the highest priority module, then (if the module does not manage any task ready to run), it asks the next high priority module, and so on. In this way, each module manages its private ready task list, and the Generic Kernel schedules the first task of the highest priority non empty module's queue.

A Scheduling Module must include all the data structures needed. It can be thought as an object in an Object oriented language; this implies that many instances of a module can be created (for example, many TBS servers with different bandwidth).

1.3.1 Module Organization

The Scheduling Modules are organized into levels, one Module for each level. These levels can be thought as priority scheduling levels (index 0 has the maximum priority).

Figure 1.4: Kernel Level Organization.

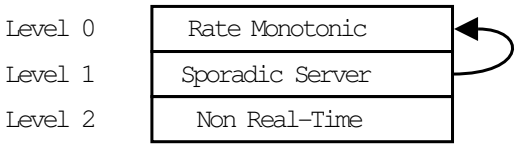

Figure 1.5: A fixed priority Module configuration.

Modules are selected for scheduling by the Model Mapper by a fixed priority strategy. When a task is given to a module, the module *owns* the task. The `task_level` field of the generic task descriptor is used to save the level index of the Module that handles the task (see Section 3.2.1).

Each Scheduling Module handles all the events that belong to its owned tasks. A task owned by a module is scheduled in background with respect to the tasks owned by a Module with higher level index. For example, in Figure 1.4, the tasks *scheduled*[2] by the XXX Scheduling Module are run in foreground; the tasks scheduled by the WWW Module run in background with respect to those of the XXX and YYY Modules, and in foreground with respect to the tasks scheduled by the ZZZ Module.

1.3.2 Sample scheduling architectures

The approach followed in the organization of the Scheduling Modules is very versatile and allows to implement different Kernel configurations. In the following, some typical scheduling architectures are described.

Fixed Priority Scheme. This is the most typical approach used in the real-time systems (see Figure 1.5). In this example, hard tasks (periodic and sporadic) are served at the highest priority, whereas aperiodic tasks are handled by a Sporadic Server. At the lowest priority level non-realtime tasks can be handled by a Round Robin scheduling policy.

Dual Priority Scheme. This configuration (described in Figure 1.6) proposes a combination of modules which were not developed to work together at implementation time. In this example, the highest priority tasks are scheduled by the RM with a Deferrable Server linked to it. Other tasks are scheduled at medium priority using EDF with a Total Bandwidth Server. At the lowest priority level, non-realtime tasks can be handled by a Round Robin scheduling scheme.. This configuration can be used to reduce the jitter of some important tasks [4].

[2]Note that the word *scheduled* is emphasized: the tasks *scheduled* by a Module are the tasks owned by the Module itself and the tasks that other modules have inserted in it.

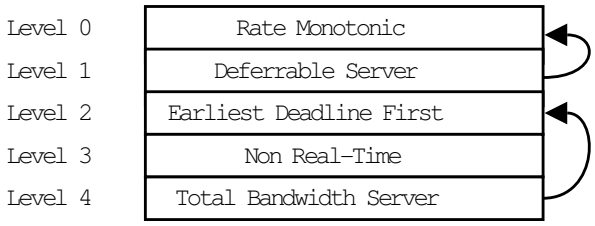

Figure 1.6: A Dual Priority Module configuration. Note that the TBS Module is put at the lowest priority level to prevent the TBS algorithm from using the background time left by other modules.

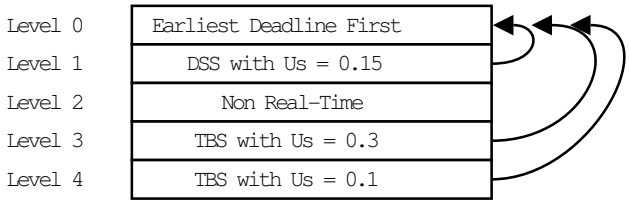

Figure 1.7: Dynamic Module Configuration.

Dynamic Multiserver. This example (described in Figure 1.7) shows how to create a complex scheduling architecture. In the example, some Hard tasks are scheduled with a set of aperiodic tasks, each one handled by a different server. Note that the EDF Module is designed to accept tasks from a generic Module, independently from the algorithm implemented by that Module. Note also that in the system there are many instances of a single module, and each instance can only manage the tasks that it owns.

Timeline Scheduling. The example illustrated in Figure 1.8 shows a Timeline scheduler integrated with a fixed priority algorithm, such as RM. Note that S.Ha.R.K. does not force the developer to use classic approaches, like prioritary task queues. The Generic Kernel does not impose any restrictions to developers.

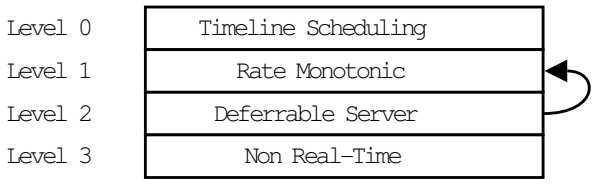

Figure 1.8: A hybrid Timeline-RM approach.

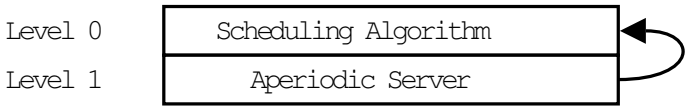

Figure 1.9: Configuration of the example in Section 1.3.6.

1.3.3 Module Interface

The interface functions provided by a scheduling module can be grouped in two classes: public and private functions. In general, a scheduling module has an interface that implements a specific behavior for each event in the system generated by a task.

In the following paragraph the various classes of functions are explained in a general way. Each function will be then described in detail in Chapter 4.

1.3.4 Public Functions

The public functions are those fuctions that are directly called by the Generic Kernel to implement the behavior of the primitives. Some of the functions are directly related to the life of a single task (e.g. task creation, task end), whereas other functions are related to the module as a whole (the scheduler, and the acceptance test).

1.3.5 Private Functions

On the other side, a module can export an interface to the public part of the same or of another module. For example, an EDF Module can export an interface (smaller than the Public Functions) that allows a generic aperiodic server to insert tasks in the EDF reqdy queue.

That is, Private Functions are called ONLY by other Public and Private Functions. They are NEVER called by the Generic Kernel.

1.3.6 An Example

In this paragraph an useful example is explained to show the use of the various functions of a Scheduling Module Interface. The interface will be described in detail in Chapter 4.

The example (see Figure 1.9) considers a configuration made using two scheduling modules, registered in the first two levels:

- at Level 0 there is a Module that implements a generic Scheduling Algorithm;
- at Level 1 there is a Module that implements a generic Aperiodic Server (that inserts his tasks into the first Module).

Then, we consider two tasks in the system:

- Task A is a task created into the Module that implements the scheduling algorithm (registered at level 0); therefore its `task_level` = 0.
- Task B is a task created into the Module that implements the aperiodic server (registered at level 1); therefore its `task_level` = 1. Moreover, the task B is inserted into the level 0 using level-0's private functions.

Both the tasks are scheduled by the level 0 public_scheduler: task A because it was created in that level; task B because it was inserted into the level 0 through the level-0's private functions.

When the scheduling procedure is called, the Generic Kernel scheduler will call the public_scheduler, stating from that of the Module Registered at level 0. In this case, the level 0 public_scheduler will choose which task really schedule between A and B. If task A is selected, the Generic Kernel will call the public_dispatch of the level 0 (because Task A's `task_level` is 0). If task B is selected, the Generic Kernel will call the public_dispatch of the level 1 (because Task A's `task_level` is 1). To handle the public_dispatch event that function will call the level-0 private_dispatch of Level 0.

1.4 Resource Modules

Resource Modules are normally used to implement some parts that do not directly interact with task scheduling, but need some information that has to be provided at task creation and termination time.

Such Modules are, for example, those that implement shared resource access protocols (they require some parameters like system ceiling, the set of mutexes used by a task, etc.), or the Modules that implements a file system that supports the specification of a disk bandwidth to be guaranteed to each task.

Like Scheduling Modules, also Resource Modules are organized into levels, one module for each level. The level number influences only the order in which events are notified to modules.

The only events that can be notified to a Resource Module are the creation and the termination of a task. At creation time an application can specify one or more Resource Models, which will be handled by the modules registered in the Kernel.

Note that the Resource Module interface is not complete, because in general a Module will need a larger interface which depends on the resource that the module itself handles. Usually, the Modules extends the interface used by the Generic Kernel, by adding a set of new functions, in a way similar to that used in an Object Oriented Language when a class is inherited from another base class.

1.5 Shared Resource Access Protocols

S.Ha.R.K. is based on a shared memory programming paradigm, so communication among tasks is performed by accessing shared buffers. In this case, tasks that concurrently access the same shared resource must be synchronized through *mutual exclusion*: real-time theory [8] teaches that mutual exclusion through classical semaphores is prone to *priority inversion*. In order to avoid or limit priority inversion, suitable shared resource access protocols must be used.

As for scheduling, S.Ha.R.K. achieves modularity also in the implementation of shared resource access protocols. Resource modules are used to make resource protocols modular and almost independent from the scheduling policy and from the others resource protocols. Each resource module exports a common interface, similar to the one provided by POSIX for mutexes, and implements a specific resource access protocol. A task may also require to use a specified protocol through a resource model.

Some protocols (like Priority Inheritance or Priority Ceiling) directly interact with the scheduler (since a low-priority task can inherit the priority from a high-priority task), making the protocol dependent on the particular scheduling algorithm (see Figure 1.10). Although a solution based on a direct interaction between the scheduler and the resource protocol is efficient in terms of runtime overhead, it limits the full modularity of the kernel, preventing the substitution

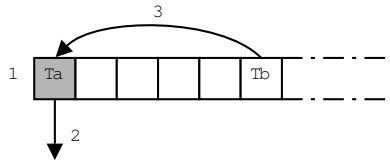

Figure 1.10: Priority inheritance implemented with an out of order queue insertion: (1) Ta blocks when it tries to access a resource; (2) Ta goes in the blocked queue; (3) Tb replaces the position of the high priority task.

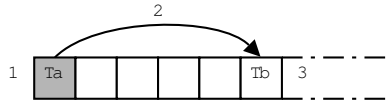

Figure 1.11: Priority Inheritance implemented through shadows: (1) Ta blocks when it tries to access a resource; (2) Ta indicates a shadow task and remains into the ready queue; (3) Tb is scheduled in place of Ta.

of a scheduling algorithm with another one handling the same task models (for example, Rate Monotonic could be replaced by the more general Deadline Monotonic algorithm).

To achieve full modularity, the S.Ha.R.K. Generic Kernel supports a generic priority inheritance mechanism independent from the scheduling modules. Such a mechanism is based on the concept of *shadow tasks*. A shadow task is a task that is scheduled in place of the task selected by the scheduler. When a task is blocked by the protocol, it is kept in the ready queue, and a shadow task is binded to it; when the blocked task becomes the first task in the ready queue, its binded shadow task is scheduled instead. In this way, the shadow task executes as if it "inherited" the priority of the blocked tasks, but no inheritance takes place, thus decoupling the implementation of the scheduling algorithm from that of the shared resource access protocol (see Figure 1.11).

To implement this solution, a new field `shadow` is added to the generic part of the task descriptor. This field points to the shadow task (see Figure 1.12). Initially, the shadow field is equal to the task ID (no substitution; see Figure 1.13). When a task blocks on a shared resource, its shadow field is set to the task ID of the task holding that resource (see Figure 1.14). In general, a graph can grow from a blocking task (see Figure 1.15). In this way, when the blocked task is scheduled, the blocking (shadow) task is scheduled, thus allowing the scheduler to abstract from the resource protocol. This approach has also the benefit of allowing a classical deadlock detection strategy: cycles have to be searched in the shadow graph when a `shadow` field is set.

Using this approach, a large number of shared resources protocols can be implemented in a

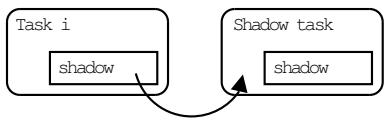

Figure 1.12: Meaning of the Task Descriptor's `shadow` field.

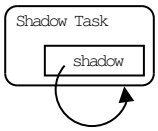

Figure 1.13: Typical scenario with no substitution.

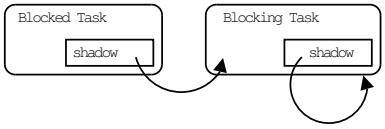

Figure 1.14: In this scenario a blocked task waits for a resource; the blocking task inherits its priority.

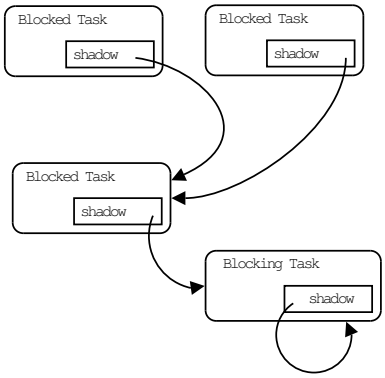

Figure 1.15: Using the shadow mechanism a graph can grow...

way independent from the scheduling implementation. The classical approach, however, can also be pursued. In addition to this method, a classical synchronization mechanism is available (see the User Manual for informations about semaphores interfaces).

1.6 The Generic Kernel

The *Generic Kernel* implements the primitives which form the interface if with the Libraries and the User Applications.

The term Generic is used because the kernel is developed abstracting from the implementation of a particular scheduling algorithm. This means that it does not provide the concept of priority, deadline, task queues, and so on. All the peculiarities of an algorithm are encapsulated in the Modules, interacting with the Kernel.

In general Applications use the Kernel through the generic primitives, asking the Kernel to handle a task set. The Kernel tries to handle the tasks passing them to the modules according to the Task Models specified for the tasks. Any module modification that does not affect the Model interface, does not require any modification to the Kernel and the Applications.

The generic kernel divides the task set in two parts: the system tasks and the user tasks. System tasks are used to manage external devices (like keyboard, mouse, file system, etc.). User tasks are tasks that belong to the Application.

Moreover, it distinguishes between scheduling and dispatching:

scheduling is the activity by which the Generic Kernel asks a module the indication of the task to be executed;

dispatching is the activity by which the Generic Kernel orders a module the execution of a task.

The following section describes some new additions implemented in the kernel to achieve modularity. They mainly concern the task descriptor and the task states.

1.6.1 Task descriptor

To have full modularity, the Generic Kernel should not be modified when implementing a new algorithm.

For this reason the task descriptor is split in several parts: one generic part, common to all the modules, and a set of parts that are dependent from the used Modules and are local to the modules.

The generic descriptor part contains all the data used by the Generic Kernel to implement the generic primitives, like context, stack address and size, error codes, statistical information, etc.

The generic part does not contain any information like deadlines, priorities, time slices, reactivation times, etc. These data are used by specific scheduling algorithms and must be contained in the corresponding scheduling modules. In particular:

- The Generic Kernel does not implement a specific scheduling policy.
- Each instance of a module has a private memory where it can store information about the owned tasks.
- Each Module uses the information stored in the generic task descriptor plus his private information, but not the information of other modules.

State	Description
FREE	Descriptor not used
SLEEP	Default state in which a task go after creation; this is the state in whichthe task returns after a call to thetask_sleep primitive
EXE	State of the task currently being executed
WAIT_*	State in which a task is put after blocking on a synchronization-primitive

Table 1.1: Task states defined by the Generic Kernel.

State name	Description
READY	This is the classical ready state in which a task waits to be executed.
BLOCKED	This is the state of a task blocked on a semaphore.
ZOMBIE	This is the state of a Hard Periodic Task afterhis termination, when it waits the end of the period tofree his used bandwidth.
LOBBY	In some implementation of the SRP protocolthis state is used to postpone the activation of atask while the system waits that for system ceiling to become zero.
IDLE	This is the typical state in which a periodic task is putat the end of an instance, to wait for reactivation at the beginning of the next period.

Table 1.2: Examples of states defined in the Modules.

- Typically, an Aperiodic Server does not use the internal structures of the Master Module, but interacts with it using the available interface functions.

1.6.2 Task States

One of the consequences of the modular approach is that many task states are local to the modules. The Generic Kernel defines only a subset of the states in which a task can be, giving freedom to the Modules to create their private subsystem.

In particular, the Generic Kernel defines only the states described in Table 1.1.

These states must be integrated with other internal states defined into the Modules. Some typical states implemented in modules are reported in Table 1.2.

1.7 Initialization and termination of the system

Before an Application can be executed, the user has to tell the Generic Kernel the modules used by the system. This is done through an initialization function in which the specified modules are registered and initialized.

Normally, the registration creates a non-realtime task, that initializes some devices and calls the standard C startup function `main()`.

When the system ends (because a function like `sys_end()` or `sys_abort()` is called) or when the last user task terminates, all the devices are closed and all the system tasks are terminated

in a clean way.

Chapter 2

Models

In this chapter we will described in detail the Models used in the Generic Kernel. For a general description of the Models see Section 1.2.

2.1 Data structures

The approach used in the definition of the kernel data structures is similar to that used in Object Oriented Languages.

In fact, there are many situations in which the kernel needs to define data structures that can be thought as a *base class*, which will be extended by the Modules that use it. For example, the Kernel has to manage Task Models to know the stack size of a task, but it does not have to know fields like priorities, deadlines, and so on. Such data have to be stored into the task model, because they represent a QoS specification for the Module that will handle the task.

For this reason, the Generic Kernel only defines a C `struct` that contains only the information that it needs (and that is common to all the derived structures). If a Module needs to extend that base class, it creates another `struct` whose first field specifies the base class type[1], whereas the other fields extend the base class.

In this way, both the Generic Kernel and the Modules can accept a pointer to the base class to access the required information.

When the Kernel primitives are used, a pointer to a derived class is passed instead of a pointer to the base class. Because the first field of a derived class is of the base class type, the pointer passed addresses the correct memory structure (the C structures are stored using the declaration order), and the Generic Kernel can safely handle the generic part of the structure.

Viceversa, when a Module interface function receives from the Generic Kernel a pointer to a base class, it will cast explicitly the struct to the correct derived type, getting the extensions of the derived class.

As it can be seen, this approach is a way to implement single inheritance and polymorphism in the C language.

2.2 Task Models

The Task Models are implemented, as described in the previous paragraph, through extensions to a base class. In the following paragraph base class is described with the main extensions to it.

[1] In this way the derived struct will inherit the fields from the base class.

```
typedef struct {
    WORD pclass;
    LEVEL level;
    size_t stacksize;
    void *stackaddr;
    WORD group;
    void *arg;
    DWORD control;
} TASK_MODEL;
```

Figure 2.1: the `struct TASK_MODEL`.

The Task Models that are included in the official distribution of the Kernel are declared into the file `include/kernel/model.h`.

2.2.1 TASK_MODEL

This is the base structure for the Task Models. Its definition is showed in Figure 2.1. In the following paragraph each field of that structure is described.

`pclass` this field contains an identifier that represents the real (derived) type of the structure. The `pclass` fields of the derived class that are included into the official distribution are included into the file `include/kernel/model.h`. The `pclass` field is a 16 bit integer

`level` this field identifies a particular level to which the task must be inserted in preference. 0 means "all levels"[2].

`stacksize` this is the dimension (in bytes) required for the stack of the task to be created. The Generic Kernel provides allocation and deallocation of the stack of the tasks (only if the parameter `stackaddr` is not specified).

`stackaddr` this is a memory pointer that can be used as stack for the task to be created. If this parameter is specified, the `stacksize` field is ignored when a task is created. The memory pointed by `stackaddr` is not deallocated at the termination of the task (deallocation must be done by the creating task. There is no check on the dimension of the memory area pointed by `stackaddr`.

`group` the tasks in the Generic Kernel are divided into groups. The group of a task is specified when a task is created, using this field. A group is a number that is common to a set of tasks in the system. This identifier is used by the primitives `group_activate` and `group_kill` to activate or kill a set of tasks in an atomic way.

`arg` the body of a task created into the system is a function that accepts a `void *` parameter. That parameter is passed at the first activation of a task and it is specified in this field.

[2]We recall that the Scheduling Modules are organized in levels. The Task Models are used when a task is created and they are passed to the Modules to find a Module that can handle the Model. All the Modules accepts Models in which the `level` field are 0 or the level number in which the Module is registered. In this way it is possible to distinguish Task Models that are related to different Modules when there are many Modules that can accept the same Model.

`control` this field contains a set of flags that represents features of the task to be created and represents also some particular states of a running task. Flags are set by the user at creation time or they can be set by some primitives. The defined flags are:

`USE_FPU` This flag is used by the OS Lib to notify the system that the task uses in its code some floating point operation. This information is used by the OS Lib to guarantee that the FPU state is saved at each context switch.

`NO_KILL` If this flag is set the task cannot be killed through a `task_kill` primitive. This flag is also used at the shutdown of the kernel (see Section 7.1), and it can be specified only at the task creation.

`NO_PREEMPT` If this flag is set, the running task cannot be preempted by another task, but it can be interrupted by a device handler. In other words, the scheduler is disabled. This flag is modified by the primitives `task_preempt` and `task_nopreempt`.

`SYSTEM_TASK` If set this flag marks the task as a system task. It can be specified only at creation time. The fact that a task is or not a system task affects the system shutdown (see Section 7.1).

`JET_ENABLED` This flag can be set only at task creation time and when set specifies that the Generic Kernel must register the Job Execution Time statistics.

`TASK_JOINABLE` This flag is set if another task can wait for the task termination through the `task_join` primitive (see Section 7.9).

`STACKADDR_SPECIFIED` This flag is set by the creation primitive if the `stackaddr` field was specified. At termination time, if this flag is not set, the stack space is deallocated by the Kernel; otherwise, the stack space is left according to the POSIX standard.

`TRACE_TASK` This flag is set if the tracer has to monitor the task events.

`KILLED_ON_CONDITION` This flag is used in the implementation of the condition variables. This flag is set by the Generic Kernel if a task is killed while it is blocked on a condition variable. In this case the task must be rescheduled to reaquire the mutex linked to the condition variable.

`KILL_ENABLED` This flag is set if the cancellation (in a POSIX meaning) is or not enabled.

`KILL_DEFERRED` This flag is set if the cancellation (in a POSIX meaning) is deferred (the cancellation is asynchronous if not).

`KILL_REQUEST` This flag registers a cancellation request made with one of these primitives: `task_kill`, `group_kill`, `pthread_testcancel`.

`CONTROL_CAP` This flag is used in the Scheduling Modules to tell Generic Kernel to generate an OS Lib event when the running task terminates its capacity.

`TASK_DOING_SIGNALS` This flag is set by the Generic Kernel when the task is executing a signal handler. This flag is used only in the function `kern_deliver_pending_signals`, contained in the file `kernel/signal.c`.

`FREEZE_ACTIVATION` This flag blocks the task activations done through the primitives `task_activate` and `group_activate`.

The flag is modified by the primitives `task_block_activation` and `task_unblock_activation`.

`WAIT_FOR_JOIN` This flag is set when a task terminates, but only if a task is joinable. The flag is used to register that the task is terminated and it is waiting for someone to do a join on it to die.

`DESCRIPTOR_DISCARDED` This flag is used in the implementation of the join primitive, and it is set by the primitive `task_create` to notify that a task descriptor, whose task is terminated and is waiting for a join, has been chosen and discarded for the creation of a new task.

`SIGTIMEOUT_EXPIRED` This flag is set for the task blocked on a `sigtimedwait` primitive, when the timeout event fires.

The internal data structures of a `TASK_MODEL` should be used only into the Generic Kernel and into the Modules. For this reason the system provides a set of macros that can be used to set the required values in a base class object. In all the described macros the parameter `m` is a `TASK_MODEL`. The macro are described below:

`task_default_model(m,p)` This macro initializes the Model `m` to a default value. The `pclass` of `m` becomes equal to the `p` parameter.

`task_def_level(m,l)` This macro specifies the scheduling level `l` of Model `m`.

`task_def_arg(m,a)` This macro specifies the parameter passed to the first activation of the task initialized with Model `m`.

`task_def_stack(m,s)` This macro is used to specify the stack dimension of the task initialized with Model `m`.

`task_def_stackaddr(m,s)` This macro is used to specify a memory address pointer to be used as a stack for the task initialized with Model `m`.

`task_def_group(m,g)` This macro is used to specify the group of tasks initialized with Model `m`.

`task_def_usemath(m)` This macro is used to specify that the task initialized with Model `m` uses the mathematical coprocessor.

`task_def_system(m)` This macro is used to specify that the task that is initialized with Model `m` is a system task.

`task_def_nokill(m)` This macro is used to specify that the task initialized with Model `m` cannot be killed.

`task_def_ctrl_jet(m)` This macro is used to specify that the task initialized with Model `m` requires the monitoring of the job execution time.

`task_def_joinable(m)` This macro is used to specify that the task initialized with Model `m` can be passed as a parameter into the join primitive.

`task_def_unjoinable(m)` This macro is used to specify that the task initialized with Model `m` cannot be passed as parameter into the join primitive.

`task_def_trace(m)` This macro is used to specify that the task initialized with Model `m` is a task for which the system tracer will register some information about the events related to it.

`task_def_notrace(m)` This macro is used to specify that the task initialized with Model `m` is a task for which the system tracer will not register any information.

2.2.2 Used conventions

In normal situations the final user never needs to instantiate an object of type `TASK_MODEL`, but objects of a type derived from it. Also the macro described in the previous paragraph cannot be used directly; instead, there are similar macros redefined for the derived types.

The standard used in the inheritance of a Model from the base model are the following:

1. The name of the derived class is obtained from the name of the base class by adding a prefix. For example, the task Model for the Soft task is called `SOFT_TASK_MODEL`.

2. The first field of a derived structure is a structure with name `t` and type `TASK_MODEL`.

3. The specific parameters added to the models (e.g., period, wcet, etc.) are inserted as normal fields after the first field. These fields represent a specification of the Quality of Service required by a task to the system at creation time. The Scheduling Modules can be structured in a way that they may use only a subset of the fields of a Model[3]. This approach limits the growth of the number of Models and achieves better independence of the applications from the task models[4].

4. The macros defined for the `TASK_MODEL` struct and described in the previous paragraph must also be defined for the new model. They should be rewritten with a name derived from the old name, like the example below:

```
#define soft_task_def_level(m,l) task_def_level((m).t,l)
```

5. The new model should provide a set of private macros similar to those provided with the `TASK_MODEL` to handle the new fields of the derived structure. For example, if the new model has a field called `period` (that contains for example a reactivation period), the new macro should be similar to the one shown below:

```
#define soft_task_def_period(m,p) (m).period = (p)
```

The five rules described above simplify the definition of new models allowing the user to cut and paste the code of a similar model and to modify some prefixes.

2.2.3 Examples of Models currently integrated into the Kernel

Currently the Kernel defines a set of Task Models directly derived from the base TASK_MODEL class. In the following paragraphs they are briefly described (for their definition look in the file `include/kernel/model.h`).

`HARD_TASK_MODEL`

This Task Model can be used to model Hard Periodic and Sporadic tasks. A Hard Periodic Task[5] is a task guaranteed using an activation period and a wcet[6], whereas a Hard Sporadic Task[7] is an aperiodic task guaranteed on a minimum interarrival time and a wcet.

[3]For example, to create a Soft Task Model it is useful to insert two parameters like period and mean execution time; these parameters are mandarory for the Scheduling Module that implements CBS, whereas they are ignored by a Scheduling Module that implements a Polling Server.

[4]For example, an Application can use a Soft Task Model specifying all the Model parameters, without knowing which Scheduling Module is really used, e.g. CBS or Polling Server).

[5]The `periodicity` field of the model must be set to `PERIODIC`.

[6]Worst Case Execution Time.

[7]The `periodicity` field of the model must be set to `APERIODIC`.

The Model allows the specification of a relative deadline. If not specified, the deadline is assumed to be equal to the next reactivation time (as in the classical task model proposed by Liu and Layland [7]).

The Model also allows a release offset to be specified. This means that any activation should be delayed by the given offset.

A Module that accepts tasks with this Model can raise the following exceptions:

`XDEADLINE_MISS` This exception is raised if the task misses a deadline.

`XWCET_VIOLATION` This exception is raised if the task tries to use more CPU time than declared.

`XACTIVATION` This exception is raised if a sporadic task is activated with a frequency greater than that declared.

`SOFT_TASK_MODEL`

This Task Model is used to specify the QoS required by a soft task. A soft task is a periodic or aperiodic task, which is handled by a server with a given (guaranteed) bandwidth (i.e., which allocates a budget Qs every interval Ts).

A soft task can specify, using a specific field, if it wants to save or skip pending activations. A pending activation occurs when a task is activated before the end of its current instance[8]. The Scheduling Modules can choose whether to handle or not this situation. Pending activations influence the behavior of the `task_sleep` and `task_endcycle` primitives.

The Soft Task Model has also a field which contains a wcet. This field can be required by some algorithm that requires this information (for example, the TBS algorithm.

Usually, the Modules that accept the soft task model do not raise any exception.

`NRT_TASK_MODEL`

This Task Model is typically used to support non real-time computations performed by tasks without temporal requirements.

Typical Modules that use these Models are Modules that implement scheduling algorithms like Round Robin, Proportional Share, priority scheduling, and POSIX scheduling.

The model has also a field to specify whether a Task have to save or skip pending activations.

Finally, the Model has two other fields (`inherit` and `policy`) used in the implementation of the POSIX scheduling algorithm [9].

`JOB_TASK_MODEL`

This Task Model is normally used by an Aperiodic Server to pass a task to a Master Module. It is not explicitly used in the Application code.

This Model extends the Base Model with a deadline and a period. A Job is a single a task instance which starts and stops without synchronization points in between.

Typically, an aperiodic server inserts a Job in the Master Module to ask for service. When that task ends its instance or it blocks on a synchronization variable, the Job dies and it is newly recreated when the task is reactivated or when it resumes from blocking.

[8] It may happen when the reserved bandwidth is less than the maximum task's utilization.
[9] This is one of the (rare) cases in which the Task Model depends on a specific Scheduling Module.

There are no fields that specify computation times (i.e., wcet, met) because the Aperiodic Servers that use that Model generally handle a budget.

The Model contains another field that specifies whether a Master Module should raise a deadline exception when a deadline is reached and the Job is still alive.

2.3 Resource Models

The resource Models are implemented in a way similar to the Task Models. The difference between Task Models and Resource Models is that it is not possible to group a set of fields common to all Resource Handlers. For this reason, the C structure that represents the base class of a Resource Model contains only a field that provides information about the real type of a Resource Model. As done for the Task Models, that field is called `rclass`.

The Resource Models available in the kernel are used in the implementation of the Priority Ceiling Protocol and the Stack Resource Policy.

In the case of Priority Ceiling, the Resource Model only adds a static priority of the task to be created.

In the case of SRP, the Resource Model only adds the definition of the Preemption Level of the task.

The Resource Models contained into the official distribution of the Kernel are included into the file `include/kernel/model.h`.

2.4 Mutex attributes

The protocol used by a mutex is decided when the mutex is initialized (at run time).

To implement the mutex initialization in a modular fashion we derived a structure, called `mutexattr_t`, from a base structure similar to that used in the Resource Models

We derived a set of mutex attributes to be used in the initialization of a mutex (a mutex is contained in a `mutex_t` type, which is similar to the POSIX's `pthread_mutex_t`).

The Mutex Attributes are declared into the file `include/kernel/model.h`.

Chapter 3

Kernel Internals

Since the Generic Kernel does not implement any memory protection, all the Modules have access to the internal data structures used for scheduling. Although the Modules could modify these data structures, typically they do not need to do so.

3.1 Kernel types

The Generic Kernel defines a set of primitive data types, that are briefly described below:

PID This type is used to contain a task index. It is an integer pointing to an entry of the task descriptor table. This type can have values in the range [0...MAXPROC-1], plus an invalid value, NIL (that is, -1).

IQUEUE This type is used to implement the task queues typically used in scheduling operations. For more information see Section 3.6.5.

TASK This type is simply a redefinition of the `void *` type and it can be used for readiness ad the returned value in the task function declaration.

LEVEL, RLEVEL They are used to index the Module descriptor Tables. They are integers and they point to a particular entry. The values of these types are [0...`MAX_SCHED_LEVEL-1`] for the `LEVEL` type and [0...`MAX_RES_LEVEL`] for the `RLEVEL` type.

bandwidth_t This type is used to store a real number in the range [0,1]. It is a 32 bit unsigned integer and its value is interpreted as $\frac{x}{MAX_BANDWIDTH}$.

task_key_t This is an integer type and it is used as an index for task specific data (similar to POSIX's Thread Specific Data).

3.2 Descriptors

In this section we present all the descriptors defined in the Generic Kernel. If not specified, these data structures are defined in the `include/kernel/descr.h` file and they are used (usually as arrays) in the `kernel/kern.c` file.

3.2.1 Task Descriptor

The `proc_des` structures used to define task control blocks[1]. In the following paragraphs the fields of the task descriptor are described[2].

DWORD task_ID Progressive number assigned to a task at its creation.

LEVEL task_level This field points to the Module that owns the task. The Generic kernel uses this field to redirect the calls to the Module owning the task.

CONTEXT context This field contains an index in the OS Lib context array that handles the context of a task.

BYTE *stack This field is a pointer to the memory used as a task stack.

TASK (*body)() This field is the pointer to the first instruction of a task body, and it is used at initialization time.

char name[MAX_TASKNAME] This is a symbolic name, whose length is defined by the `MAX_TASKNAME` constant.

WORD status This is the task status.

WORD pclass This field is the Class code of the Task Model used during task creation. It can be used by Modules to know the typology of the task (useful if the task supports many types of Models; see Section **??**, `level_accept_task_model` function).

WORD group This is the task group. The value 0 is used to index a single task. This field is not used as a GID in classical Unix systems but is used to kill or activate a group of tasks in an atomic way.

WORD stacksize Stack dimension (in bytes).

DWORD control Task Status flags. A description of the bits in this field is reported in Section2.2.1.

int frozen_activations Number of frozen activations. Useful only if the `FREEZE_ACTIVATION` flag of the `control` field is active (see Section 7.4).

int sigmask This is the blocked signal task mask (see `kernel/signal.c` and also [9]).

int sigpending This is the pending signal task mask (see `kernel/signal.c` and also [9]).

int sigwaiting This is the mask of the signals on which a task is waiting, blocked on a `sigwait` primitive (or similar).

int avail_time This field contains the remaining computation time of a task (see Section 3.5).

PID shadow This is the shadow pointer (see Section 1.5).

struct _task_handler_rec *cleanup_stack Pointer to the first element of the cleanup stack.

[1]The name of this structure is derived from the previous versions of Hartik. However, it is a *task* descriptor and not a *process* descriptor.

[2]The following fields have been removed from the structure in the last releases: request_time, priority, timespec_priority, prev, next.

int errnumber This is an error number local to the running task. The symbol `errno` defined into the C standard is implemented as a macro that refers to this field of the running task.

TIME jet_table[JET_TABLE_DIM] This table contains the computation times consumed by the last JET_TABLE_DIM instances of the task. The table is handled as a circular buffer.

int jet_tvalid This is the number of valid items in the `jet_table` table.

int jet_curr This is the current element in the `jet_table` table.

TIME jet_max This field contains the maximum time (in microseconds) consumed by a task, among all its instances.

TIME jet_sum This field accumulates the execution times (in microseconds) of all task instances from the start of the task (or from the last call to `jet_delstat`). This field, together with the `jet_n` field, is used to compute the mean execution time of a task instance.

TIME jet_n This field contains the number of instances involved in the computation of `jet_sum`.

PID waiting_for_me This field contains the identifier of the task which is currently blocked for a `task_join` on the current task. The field value is NIL if no task is blocked for a join on it.

void *return_value This is the value returned by a task when it dies. This value is memorized if the task is joinable, waiting for someone to synchronize on it with a join.

void *keys[PTHREAD_KEYS_MAX] This array contains the task specific data.

struct condition_struct *cond_waiting This field is NULL or it points to the condition variable on which the task is blocked[3].

int delay_timer This field is used in the implementation of the blocking primitives with timeout. Usually this field contains the OS Lib index of the event created for the wake-up.

int wcet This field can be used by Modules to store some temporal information. It is generally used in conjunction with the `avail_time` field.

3.2.2 Level descriptor

In this Section we describe the fields contained in the `level_des` structure, that implements the Scheduling Module Descriptor. These Modules will fill the proposed interface, defining also all the interface functions. If a Module does not implement an interface function, it redefines that function in a way that it will raise an exception if called. The fields defined for the `level_des` structure are the following:

Private Functions, Public Functions The rest of the interface is made of a set of function pointers that implements the *virtual* functions of the Scheduling Modules. That functions are described in the Chapter 4. All that functions have as first parameter a LEVEL field that can be used to find the Module Descriptor and then the private structures of that Modules[4].

[3]This field is present because if a task is killed during a block on a condition wait, the POSIX standard requires that the task reacquires the mutex linked to that condition before it dies.

[4]The behaviour of the LEVEL parameter passed to that function is the same of the hidden paramether *this* of many Object Oriented languages.

Interval reserved to	Codes
Generic Kernel	[0...MODULE_STATUS_BASE-1]
Scheduling Algorithm	[MODULE_STATUS_BASE ...APER_STATUS_BASE-1]
Aperiodic Servers	[APER_STATUS_BASE ...LIB_STATUS_BASE-1]
Others (Resource Handling, libraries, etc.)	>= LIB_STATUS_BASE

Table 3.1: Partitioning of the codes for the status field of the task descriptor.

3.2.3 Resource module descriptor

In this Section we describe the fields contained in the `resource_des` structure, which implements the Resource Module Descriptor. These Modules will fill the proposed interface, defining all the required functions. If a Module does not implement an interface function, it redefines that function in a way that it will raise an exception if it is called. The fields defined for the `resource_des` structure are the following:

char res_name[MAX_MODULENAME] This filed contains a symbolic Name of the Module, for statistical purposes. The file `include/modules/codes.h` contains the name of the Modules distributed with the Kernel.

WORD res_code This field contains a numeric identifier that identifies a Module in an unique way. The codes of the Modules distributed with the kernel are written in the file `include/modules/codes.h`.

BYTE res_version This is a version number for the Module. The version numbers of the Modules distributed with the Kernel are reported in the file `include/modules/codes.h`.

int rtype This field is used to identify the extended interface implemented by a Resource Module. This field is necessary to implement some primitives that use a particular extended interface, that need to know which of the resource modules registered in the kernel have a particular extension.

Resource Calls The rest of the interface is made of a set of function pointers that implement the *virtual* functions of the Resource Modules. These functions are described in Chapter 5. All these functions have as first parameter a RLEVEL field that can be used as the similar parameter of the Scheduling Module virtual functions.

3.3 System states

As mentioned in Section 1.6.2 the concept of task state has a local meaning, and it is stored in the `status` field of its task descriptor.

The values that the status field can have are divided in four intervals, described in Table 3.1. Each Module Should use the codes in the correct range of values.

This approach allows to handle many Module configurations. The fact that two Scheduling Modules use the same status code for different meanings is not a problem, because the codes are used internally to the Modules.

3.4 Kernel Global Variables

The following global variables are defined in the Generic Kernel and can be used by the Modules.

proc_des proc_table[] This variable is the task descriptor array. Not all the entries of this array are used. The descriptor really allocated are handled by the `task_create` primitive.

level_des *level_table[] This variable is the array that memorizes a set of *pointers* to the Scheduling Modules Descriptors. When a Module registers itself into the Kernel, an entry of this array is allocated, and it points to an *extension* of the `level_des` descriptor type. In an object oriented interpretation it can be thought as a polymorphic Module array (see Section 2.1).

resource_des *resource_table[] This variable memorizes an array of pointers to extension of the resource_des structure. For this variable the same comments for the level_table apply.

PID exec This variable contains the task returned by the Scheduling Module scheduling operation. Typically this variable is used only into the `scheduler()` function of the Generic Kernel. This variable cannot be modified, it can only be read and checked with the field `exec_shadow`. This variable has the value `NIL` (-1) when a primitive blocks the running task, and the scheduler is not yet called. This variable *does not* point to the running task. Use `exec_shadow` instead!

PID exec_shadow This variable points to the running task. It can be different from the value contained into `exec` because of the Shadow mechanism (see Section 1.5). This variable cannot be modified, it can only be read and check with the field `exec`. This variable has the value `NIL` (-1) when a primitive blocks the running task, and the scheduler is not yet called. When a reference to the running task is needed, this variable has to be used.

int cap_timer This variable is different from -1 only if the Generic Kernel or a Module created a capacity event. This variable can be used by the Scheduling Modules that do not utilize the `CONTROL_CAP flag` for their tasks (see Section 3.5).

struct timespec schedule_time This variable contains the system time in which the scheduler was called the last time. The first thing the Kernel function `scheduler()` does is the setting of that variable. This variable is also used as finish time for an execution time interval.

struct timespec cap_lasttime This variable is a copy of the `schedule_time` variable, done at each call of the `scheduler()` function. The value of that variable is used as the start time of the last interval executed by a task. It is used to implement the time accounting.

DWORD res_levels This variable is the number of resource module level descriptors allocated. It is modified only by the `resource_alloc_descriptor` function (see Section **??**).

int task_counter This variable is a variable that counts the number of User tasks actually present into the system (see Section 7.2).

int system_counter This variable is a variable that counts the number of System tasks actually present into the system (see Section 7.2).

int mustexit This variable is used by the system primitives `sys_end` and `sys_abort` to block the context changes into the event handlers.

int calling_runlevel_func This variable is a flag. It is set to 1 when the system executes some system functions (registered through the `sys_atrunlevel()` function). This flag influence the `task_activate` primitive, because that primitive has to do different stuffs depending on the value of that variable (see the code of the primitive in the file `kernel/activate.c`).

IQUEUE freedesc This variable is the free task descriptor queue. It is handled by the Kernel and must be used by a Scheduling Module into the call `task_end` call to free a task descriptor of a task terminated correctly.

TIME sys_tick If the OS Lib is initialized with the *one shot timer* the variable contains the system tick (in microseconds). Otherwise, the one-shot timer is used, and this variable has a value of `0`.

3.5 Temporal protection

The Generic Kernel supports task temporal protection through a set of procedures and data structures that may be used by the Scheduling Modules. The generic Kernel does force to use its own functions, so a Module can define itself a policy to ensure temporal protection. In this Section the proposed functions to implement the temporal protection are described.

The Generic Kernel handles the time capacity of the tasks in the system using the creation and the deletion of a specific OS Lib event that simply reschedules the system. The end of the capacity is seen by the Modules as a normal epilogue done in the case of a preemption.

The timer event is created when a task is dispatched, and it is deleted by all the primitives that may cause preemption. In an instant there is at least one capacity event pending, that is the event of the running task.

The Generic Kernel defines for each task the bit `CONTROL_CAP` that is memorized into the field `control` of the task descriptor. That bit is set by the Scheduling Modules to require for a task the creation of the capacity event at dispatch time.

The Scheduling Modules have also to check the capacity exhaustion each time a task is descheduled.

Capacities are handled using the following variables provided by the Kernel:

- The `wcet` field of the task descriptor usually memorizes a characteristic time for a task (i.e., it is used to store a worst-case execution time or a mean execution time). It is included in the task descriptor to avoid the allocation of such an information for all the modules. It is *not* used by the Generic Kernel.

- The `avail_time` field of the task descriptor is used to handle the available execution time for a task. The capacity event generated by the Generic Kernel at dispatch time has an activation time that is equal to `schedule_time + avail_time`. The field is updated by the generic kernel each time a task is descheduled, decrementing the slice just executed by the task.

- The `cap_timer` field is used by the Generic Kernel to know whether a capacity event is or not pending. If the field is equal to -1, there are no events, otherwise the value is the index of the capacity event. If the field is not equal to -1 the Generic kernel will delete automatically the event each time the system is rescheduled. The Generic Kernel does not check who has created the event pointed by `cap_timer`, so a Scheduling Module that not use the flag `CONTROL_CAP` can create a capacity event and put his index into `cap_timer`, avoiding the removal of the event directly in the code of the Module.

- The `cap_lasttime` field contains the time of the previous reschedule of the system. The accounting of the computation times is done from the difference between `schedule_time` and `cap_lasttime`. The time used by the Kernel primitives is accounted in the running task.

The functions called by the capacity event have only to reschedule the system. The Scheduling Modules that have to generate themselves the capacity event can store their events in the `cap_timer` filed if they want an event to be removed at each preemption. The events can call the Generic Kernel Function called `capacity_timer` (it does not accept any parameter).

3.5.1 Negative capacities

The contents of the `avail_time` field can be negative. This fact, theoretically impossible, can happen for two reasons:

- first, the OS LIb event handler can not guarantee the delivery of a capacity event at exact time (a small delay of some microseconds may occur). So, a capacity event may be delivered after the capacity is exhausted, so that the task capacity becomes negative. We can suppose that negatives values are smaller than the usual time for the tasks;
- second, because the shadow mechanism can execute a task that has finished his budget; that task will not be descheduled until the shadow of the higher priority task points to it. This behavior is correct because we want that the blocking task will ends its critical section to limit the blocking time of the higher priority tasks. This approach is supported by some theoretical results (see [6]).

Scheduling modules can use two strategies to cope with the problem given by negative capacities.

The first solution is to recharges the capacity with an amount equal to Q_s; if the capacity is still negative, the task will "loose a turn". This solution is good for algorithms that provide a periodic replenishment (like for example Round Robin, Polling Server or Deferrable Server) but not for algorithms like Sporadic Server or Constant Bandwidth Server, where the replenishment is due to the capacity usage (in these algorithms the server would not become active).

The second solution is to replenish the budget up to Q_s; this solution is good for all types of algorithms, but we have to consider that the server utilization factor becomes $\frac{Q_s+\Delta}{T_s}$.

3.6 Utility functions

This section contains a set of utility functions that can be used to simplify the writing of the Modules. Often these functions are simple redefinitions of the OS Lib functions. Use these functions instead of those provided by the OS Lib.

3.6.1 Event Handling

The Generic Kernel uses the Generic kernel event handling provided by the OS Lib. The following functions should be used (see `include/kernel/func.h`):

- `kern_event_post` can be used to post an event. The redefinition adds a check on the index, raising an exception if the OS Lib event queue is full.
- `kern_event_delete` can be used to remove an OSLIB event. The unique parameter of the funcion is the eveni ID returned by the kern_event_post function.

```
void *kern_alloc(DWORD s);
void *kern_alloc_aligned(size_t size, lmm_flags_t flags,
      int align_bits, DWORD align_ofs);
void *kern_alloc_gen(size_t size, lmm_flags_t flags,
      int align_bits, DWORD align_ofs,
      DWORD bounds_min, DWORD bounds_max);
void kern_free(void *block, size_t size);
void *kern_alloc_page(lmm_flags_t flags);
void kern_free_page(void *block);
void *DOS_alloc(DWORD size);
void DOS_free(void *ptr,DWORD size);
```

Figure 3.1: Memory allocation functions.

- `event_need_reschedule`, with no parameters. It has to be called into the event handlers that have to reschedule the system. The Modules never have to call the schedule function into event handlers.

3.6.2 Exception Handling

The Generic Kernel exception handling is based on the real-time signal interface of the POSIX standard. In particular, the exceptions are implemented with the signal number 9, SIGHEXC, that typically writes a message to the console and ends the system.

The function to use to raise an exception is:

```
void kern_raise(int n, PID p);
```

As a result of this function call, a real-time signal is enqueued into the system and the n parameter is passed into the field `si_value` of the structure `siginfo_t` passed as a parameter to the signal handler.

3.6.3 Memory Management

The Generic kernel provides a memory allocator based on Flux OS-Kit LMM[5]. The LMM is in the `kernel/mem` directory; the functions defined in this subsection are in the file `kernel/mem.c`. These functions must be called with interrupt disabled.

The allocator divides the memory into three regions that can be specified with some #defines into the flag parameter of some functions:

- Memory addresses below 1 Mb (flag field set to `MEMORY_UNDER_1M`)
- Memory addresses from 1 Mb to 16 Mb (flag field set to `MEMORY_FROM_1M_TO_16M`)
- Memory addresses below 16 Mb (flag field set to `MEMORY_UNDER_16M`)
- Memory addresses over 16 Mb (flag field set to `0`)

Memory management functions are described below (see also Figure 3.1):

`kern_alloc` This function allocates a memory block of s bytes returning a `void *`. All the addresses given are supposed to be good for the block. The function returns `NULL` if there is not a free memory block with the required size.

`kern_alloc_aligned` This functions allocates an aligned memory block. The first `align_bits` of the block must be equal to the lowest `align_bits` of `align_ofs`. In other words, `align_bits` specifies an alignment as a power of 2, whereas `align_ofs` is a natural offset. The function returns `NULL` if there is not a block with the required characteristics.

`kern_alloc_gen` This function allocates an aligned memory block in a way similar to `kern_alloc_aligned`, with the additional condition that the block must be found within the addresses `bounds_min` and `bounds_max`.

`kern_free` This function will free a block allocated with one of the previous functions. Note that this function requires the dimension of the block to allocate.

`kern_alloc_page` This function allocates an aligned page of memory (4 Kb).

`kern_free_page` This function frees a page allocated with the `kern_alloc_page` function.

`DOS_alloc, DOS_free` These functions allocate memory under the first Mb.

3.6.4 Context switch

The creation and the deletion of a context in the Generic Kernel is made using the OS Lib functions, properly renamed into `kern_context_create` and `kern_context_delete`.

To change context, the Generic Kernel provides two functions, `kern_context_save` and `kern_context_load`. First function can be used to start a kernel primitive (disabling interrupts, whereas the second can be used to enable the interrupts, change the context using the OS Lib functions, dispatch pending signals and test for asynchronous cancellation. Note that currently the kernel primitives run on the same stack of the caller thread, and the primitives are simply called disabling the interrupts.

These functions must never be called into the standard interface of the scheduling modules. When a OS Lib event need to reschedule the system because there is a preemption, the function `event_need_reschedule` must be called instead.

The functions described in this paragraph are into the `include/kernel/func.h` file.

3.6.5 Queues, arrays and pointers

To simplify the use of the internal Kernel references, the following approach is used:

- Whenever possible, the descriptor arrays are statically allocated.
- Each descriptor is identified by an integer which represents the index in the descriptor array (e.g. PID, LEVEL, RLEVEL).

To handle task queues, the Generic Kernel provides some utility functions which can speed up the Module writing; however a module can use its own functions to enqueue tasks.

The implemented functions do not use any fields of the task descriptor; they are implemented with a double linked list. The prototypes of the functions for queue management are shown in Figure 3.2. They are into the file `kernel/iqueue.c`, and `include/kernel/iqueue.h`, described below.

```
#define IQUEUE_NO_PRIORITY 1
#define IQUEUE_NO_TIMESPEC 2
struct IQUEUE_shared {
  PID prev[MAX_PROC];
  PID next[MAX_PROC];
  struct timespec *timespec_priority;
  DWORD *priority;
};
typedef struct {
  PID first;
  PID last;
  struct IQUEUE_shared *s;
} IQUEUE;

void iq_init (IQUEUE *q, IQUEUE *share, int flags);
void iq_priority_insert (PID p, IQUEUE *q);
void iq_timespec_insert (PID p, IQUEUE *q);
void iq_insertfirst (PID p, IQUEUE *q);
void iq_insertlast (PID p, IQUEUE *q);
void iq_extract (PID p, IQUEUE *q);
PID iq_getfirst ( IQUEUE *q);
PID iq_getlast ( IQUEUE *q);
PID iq_query_first(IQUEUE *q);
PID iq_query_last(IQUEUE *q);
struct timespec *iq_query_timespec(PID p, IQUEUE *q);
DWORD *iq_query_priority (PID p, IQUEUE *q);
PID iq_query_next (PID p, IQUEUE *q);
PID iq_query_prev (PID p, IQUEUE *q);
int iq_isempty (IQUEUE *q);
```

Figure 3.2: Prototypes of the task queue handling functions.

Basically, an IQUEUE has an "I"nternal prev/next structure, that may be shared between one or more queue. Of course, the user MUST guarantee that the same task will not be inserted in two IQUEUEs that share the same prev/next buffer.

Internal queue initialization:

share = &x the internal data structure of the IQUEUE x is used to enqueue the tasks.

`share = NULL` an internal data structure to handle prev/next pairs is dynamically allocated (The amount of memory that is allocated can be reduced using the flags).

flags can be used to reduce the memory usage of an IQUEUE when `share=NULL`:

`IQUEUE_NO_PRIORITY` the iqueue do not provide internally a priority field

`IQUEUE_NO_TIMESPEC` the iqueue do not provide internally a timespec field

note that, if these flags are used, the corresponding insert functions will not work!. The default value for the flags is, of course, 0.

The queue insertion is made by the following functions:

iq_insert insertion based on the `priority` field.

iq_timespec_insert same as above but use the `timespec_priority` field

iq_insertfirst insert in the first position of the queue

iq_insertlast insert in the last position of the queue

The queue extraction functions: basically extracts a task p from the queue q. There are three versions of the function:

iq_extract extracts given a task p (that must be inserted in the queue);

iq_getfirst extracts the first task in the queue, NIL if the queue is empty;

iq_getlast extracts the last task in the queue, NIL if the queue is empty;

Seven queue query functions are also provided. The first two functions (query first/last) return the first and the last task in the queue, NIL if the queue is empty. The second two functions (query priority/timespec) can be used to get/set the priority or the timespec field used when queuing. The third two functions (query next/prev) can be used to scan the queue elements. The last function can be used to test if a queue is empty.

3.6.6 Initialization functions

The Generic Kernel supports the specification of the functions to be called at system initialization and termination (see Section 7.2). These functions can be registered through the following system primitive:

```
int sys_atrunlevel(void (*f)(void *),void *p, BYTE when);
```

The parameters for that function are:

`f` the function to be registered;

`p` the parameter to be passed to function `f` when the function will be called;

`when` is the situation in witch that function will be called. The correct values are the following:

`RUNLEVEL_INIT` The function will be called after Module registration, when the system is just entered in multitasking mode but no thread executed yet;

`RUNLEVEL_SHUTDOWN` The function will be called after a call to `sys_abort` or `sys_end`; The system is still in multitasking mode;

`RUNLEVEL_BEFORE_EXIT` The function will be called when the Kernel exits from multitasking mode;

`RUNLEVEL_AFTER_EXIT` The function is called before the system hangs (or returns to the host OS, if the proprietary extender is used).

It is also possible to specify with an OR operator a flag `NO_AT_ABORT` that disables the call to the functions if the system is exiting with a `sys_abort` function.

You can post at most `MAX_RUNLEVEL_FUNC` functions.

3.6.7 Interrupt disabling, printf and system termination

The Generic Kernel redefines the following OS Lib functions (see the `include/kernel/func.h` file):

`cli` renamed in `kern_cli`

`sti` renamed in `kern_sti`

`ll_fsave` renamed in `kern_fsave`

`ll_frestore` renamed in `kern_frestore`.

To display some messages to the console, the Generic Kernel provides two functions called `printk` and `kern_printf`. These functions are similar to the standard C `printf` function, except that they write directly to the console. The `printk` function allows to specify a kind of "importance" for the message that the function displays. For more information look at the file `include/kernel/func.h` and `include/kernel/log.h`.

Finally, a clean system termination can be obtained through the `sys_end` and `sys_abort` primitives. For more information look at the file `include/kernel/func.h` and read Section 7.2.

Chapter 4

The Scheduling Modules

This chapter describes the interface of a generic Scheduling Module, the semantic of the functions that compose the interface, and a set of conventions used to write the Modules. For more information on the Scheduling Module Architecture see Section **??**.

4.1 Task Lifecycle

To understand the interface of a Scheduling Module, we present a simple view of the events that refer the life of a task, from its creation to its end. The Module Interface reflects these events.

Figure 4.1 illustrates a simple case, in which a task is created, activated, and then preempted by another task that dies after a while. The following events are generated:

create This event is generated at task creation. The Scheduling Module initializes the data structures for the task activation.

activate This event is generated when a task is explicitly activated by a call to a user primitive. That event authorizes the Scheduling Module to insert the task in the set of the schedulable tasks.

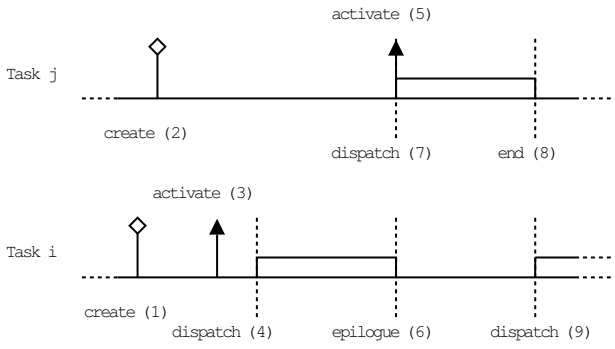

Figure 4.1: A simple scenario: tasks i and j are created and activated. Task i is executed, and, after a while, it is preempted by task j. The numbers in parenthesis denote the event sequence.

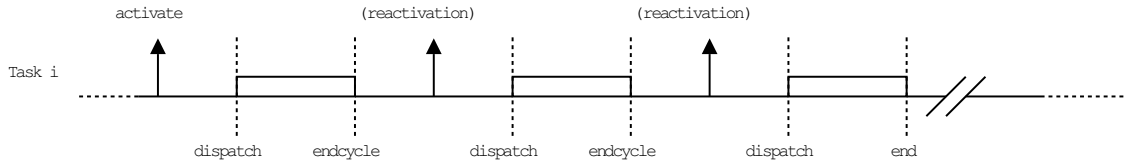

Figure 4.2: Activation handling: a periodic task is activated once, and it executes for three activations .

dispatch This event is generated when a task, after it has been scheduled[1], is actually executed. The Scheduling Module updates the data structures to register the execution of the task.

epilogue This event is generated all the times a task is preempted by another task[2]. The Scheduling Module usually reinserts the task in the set of the schedulable tasks.

end this event is generated when a task ends its execution. The Scheduling Module is authorized to free the task descriptor and any data allocated for the task.

Note that the POSIX standard does not differentiate between task creation and task activation; in fact, the `pthread_create` primitive creates and activates directly a task.

Figure 4.2 shows a sample sequence of events produced by a periodic task. A task is activated, it executes for three instances and then it falls asleep waiting for another explicit activation. The new events introduced in this scenario are:

endcycle The endcycle event is the termination of the current task instance. It is generated by a primitive inserted into the task code. If the task is a periodic task, it will be reactivated by the Scheduling Module at the beginning of the next period. If the task is an aperiodic task, this event implies that the task will wait for an explicit activation (through an activate event).

(reactivation) This event is not created directly by the user with a primitive, but it is handled internally by the Scheduling Module that handles the task. This event reactivates the task, and it is usually delivered at the end of a period.

Although the event *endcycle* terminates an aperiodic or a periodic job, and it can differ in the way they handle pending activations. A *pending activation* is an *activate* event which is delivered when the task has not ended the previous instance. The handling of pending activations is left to the Scheduling Modules.

Figure 4.3 shows the *block* ed *unblock* events. These events are generated to handle the behavior of a synchronization primitives, which generally blocks a task (*block* event) and then activate again the task after a while (*unblock* event). The events behave as follows:

block It disables the task scheduling because the task is arrived at a synchronization point.

unblock This event is used to notify to the task that the synchronization is occurred, so the task scheduling must be enabled.

The Generic Kernel guarantees that an *unblock* event will never be called before a corresponding *block* event. That is, these two events are coupled.

[1]The scheduling event is not showed in the figure.
[2]This event is also raised if the task finishes its time capacity allocated on its server (for soft tasks).

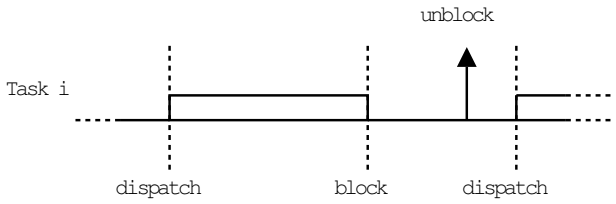

Figure 4.3: Task synchronization. A task calls a synchronization primitive, which blocks the task using an extract event. When the task will be able to continue, an insert event will be called.

4.2 Assumptions on Task Queues

The modules distributed with the Kernel use the following assumptions when managing task queues:

- The variables of type `IQUEUE` are allocated into the extensions of the level descriptor and *not* outside;
- The running task is extracted from the ready queue (if there is one in the Module) at dispatch time[3];
- A Module can handle queue and other data structures than those provided by the Generic Kernel. In this case all the data structures shall be inserted in the level descriptor extension and the functions that use them shall be visible only in that Module (for example look at the file `kernel/modules/srp.c`).

4.3 Scheduling Module Interface

This section describes the interface of the functions that have to be implemented to develop a Scheduling Module. The functions described in this Section are those represented by the dashed rectangles named *Public Functions and Private Functions* introduced early in this chapter. They are *not* user primitives, but they are called to implement the scheduling behavior of a primitive. The function names reported in this section are the name of the function pointers contained into the `level_des` structure (defined into `include/kernel/descr.h`). When a designer implements a new Scheduling Module, he writes the correspondent functions and then he sets the correct values into the correct `level_des` structure (as an example, a template application is provided on the web site). The user primitives are listed into the S.Ha.R.K. User Manual (it can be found, together with the template application, at `http://shark.sssup.it /download.html`). For more information see also at Sections **??** and 3.2.2.

All the interface functions have as first parameter a variable of `LEVEL` type, used to obtain a pointer to the level descriptor of the current Scheduling Module; some functions may also have additional parameters.

We recommend that the functions listed in Table 4.1 should use the global variable `schedule_time` to get the system time, and should not use the `kern_gettime` function provided with the OS Lib.

WARNING: All the Scheduling Modules functions are called with *interrupts disabled.* They should never consume more than a few microseconds!!!

[3]A policy that leaves the running task at the head of the ready queue is also suitable.

Type	Function

Public Functions	`public_scheduler`
	`public_dispatch`
	`public_epilogue`
	`public_message`
	`public_block`

Private Functions	`private_dispatch` `private_epilogue`

Table 4.1: These functions should use the variable `schedule_time` to read the current time.

4.3.1 Public Functions

These functions are directly called by the Generic Kernel to implement the behavior of a scheduling algorithm.

```
PID (*public_scheduler)(LEVEL l);
```

This function is the scheduler of the current Scheduling Module.

It must return the scheduled task among those handled by the Module[4].

The scheduled task must be selected only using the private data structures of the Module, *prescinding from the other Modules registered in the system.* The fact that a task is returned by this function to be scheduled does not imply that that task will be executed (dispatched) immediately[5]!

So, the level scheduler shall not:

- Modify the pointer to the running task (in other words, the variables `exec` and `exec_shadow`);
- Handle timers for deadline, capacity exhaustion, and things like that;
- Set the data structures preparing the execution of the task (for example, if the Module uses a ready queue, the task must not be extracted from the queue[6]).

If the level does not implement a scheduler (because, for example, it is an Aperiodic server that inserts all its tasks in another Module[7]), or if the Module currently does not have any ready task, the returned value must be `NIL`.

```
int (*public_guarantee)(LEVEL l, bandwidth_t *freebandwidth);
```

This function implements the on-line acceptance test. This function should only consider the tasks directly inserted in the module, and it does not consider the tasks inserted in the module through private functions (their guarantee is made by the Module that owns them).

The function is called with an additional parameter that is the free bandwidth left by the Modules with level number less than `l`. The acceptance tests that can be implemented are those based on the Utilization Factor.

[4]That is, the tasks owned by the Module plus the tasks that other Modules inserted in it using the private functions.

[5]Look at section **??**

[6]This will be done at dispatch time!

[7]look at `kernel/modules/cbs.c`

The function returns 1 if the current task set can be guaranteed using the free bandwidth available, 0 otherwise. The `freebandwidth` parameter must be decreased by the function by the amount of bandwidth used by the task being guaranteed.

If the pointer to this function is registered in the `level_des` descriptor of the Module with a NULL value, the acceptance test procedure will stop and the whole task set is considered guaranteed (see Section 7.3).

This function is called by the Generic Kernel each time a task is created in the system. The call is issued after the task is created using the task call `public_create` and after the Resource Models of the new task have been registered. The public_guarantee functions are called starting from level 0 until a NULL pointer is reached or the task set cannot be guaranteed.

```
int (*public_create)(LEVEL l, PID p, TASK_MODEL *m);
```

This function is called by the Generic Kernel into the `task_create` primitive to create a task into a Module. The function has two additional parameters: the first is the task descriptor allocated by the Generic Kernel for the new Task, and the second is the Task Model passed at creation time.

The function returns 0 if the Module can handle the TASK_MODEL passed as parameter (that is, if the Module can handle the pclass of the TASK_MODEL, and if the level field is 0 or l), -1 otherwise.

The functions must set the Module internal data structures with the QoS parameters passed with the Task Model. The function does not enable the task to be scheduled in the system (i.e., if a ready queue is implemented, the new task should not be inserted in it).

The `task_create` primitive sets the task state to the default `SLEEP` value, and sets the flags of the `control` field of the task descriptor to the values given into the `control` field of the `TASK_MODEL` structure. These settings can be modified by the public_create function.

The acceptance test on a new task is called after this function.

```
void (*public_detach)(LEVEL l, PID p);
```

This function is called into the Generic Kernel `task_create` primitive when an error is occurred during the creation of a new task.

The function receives as an additional parameter the task identifier passed to the `public_create` function. This function is called only after public_create, and it must reset the data structures allocated internally by the current Module during public_create.

```
void (*public_end)(LEVEL l, PID p);
```

This function implements task termination. When this function is called the task has been killed by someone (e.g., task_kill() or pthread_cancel()), or it is ended. All the references to task p have to be removed from the internal data structures of the Module.

The `public_end` function is called after the POSIX's cleanup functions, after the POSIX's thread-specific data destructors, and before the destructors of the Resource Modules.

When the function is called the task is in the EXE state. The typical actions done by this function are the following:

- The task is extracted from some internal queue.
- All the pending events for the task (e.g., deadline and capacity timers) are removed.

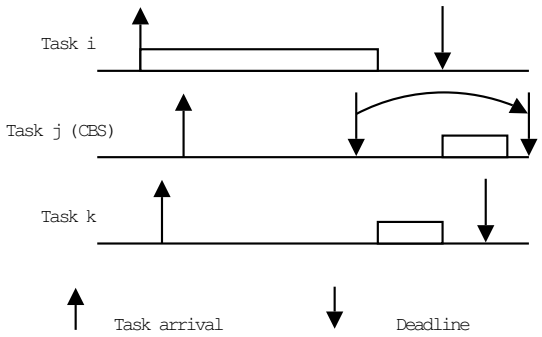

Figure 4.4: Use of the `public_eligible` function. Consider a CBS Module that inserts its task j into an EDF-NP (EDF non-preemptive) Module. EDF-NP will schedule task i first. When task i ends, task j (CBS) is scheduled. However, this task has an obsolete deadline. The `public_eligible` function called on the CBS Module when the EDF-NP Module tries to schedule task j allows the CBS Module to postpone the deadline.

- The task is inserted into the `freedesc` queue. When the task is inserted in this queue the task descriptor of the task may be reused by the Generic Kernel.

  If the Module implements some form of guarantee on the task set, the insertion of the descriptor p into the `freedesc` queue may be postponed until the bandwidth used by the task will be totally released. This may occur after task's termination (for example, in the EDF Module a guarantee is implemented, and the bandwidth is decreased at the end of the current period. Hence, the task descriptor will be inserted into the `freedesc` queue at the end of the period.

```
int (*public_eligible)(LEVEL l, PID p);
```

This function is called by the Generic Kernel into the global scheduling function (see Section 7.5) and is used to ensure the Kernel of the correctness of the value returned by the public_scheduler.

The function receives as an additional parameter that is the task returned by the `public_scheduler` call. It must return 0 if the task can be scheduled, -1 otherwise. If a -1 is returned, the system will call the level scheduler function again to choose another (may be the same) task.

This function is used when implementing aperiodic servers and the Module needs to know when its tasks will be scheduled by a level_scheduler of the Master Module, to update some old out-of date parameters.

This function is useful in pathological or unplanned situations. For example (see Figure 4.4), we have implemented a CBS module in a way that it uses another Scheduling Module. The CBS Module inserts its tasks in that module using the guest calls; after that, it waits for its tasks to be scheduled. In general, it may happen that, because of some overload condition, the CBS task will be scheduled after its deadline. Through the `public_eligible` function the CBS Module can postpone the task deadline, so causing the scheduler to be called again to manage the new situation.

```
void (*public_dispatch)(LEVEL l, PID p, int nostop);
```

This function is called by the Generic Kernel to notify a Module (registered at level l) that his task p is going to be the running task.

When this function is called it is not possible to change the task selected for execution. It is not possible to avoid the execution of a task that is on the tail of a shadow chain (see Section**??**). The `public_eligible` function should be used instead whenever a task cannot be scheduled but it is chosen by a level scheduler.

The function receives two additional parameters: the task `l` that will be executed and a `nostop` parameter. The value of the latter parameter is the result of the logic expression `exec == exec_shadow`, where the value of `exec_shadow` is computed by the `scheduler()` function (see Section 7.5) after following the shadow chain. If the value nostop is 0, no capacity event should be generated by the Module.

In practice, the `public_dispatch` can be thought as a prologue in which the Scheduling Modules set the internal data structures to allow a task to be executed. The state of the task is set to EXE before the function call. The function shall not modify the task state, as well as the `exec` and `exec_shadow` variables.

A few typical actions for this function are described below:

- the task is removed from the ready queue, if there is one;
- if the Module does not use the `CONTROL_CAP` flag (see 3.5, and kernel/modules/ps.c) but it needs capacity control it is necessary to create a capacity event with the `kern_event_post` function. Remember that a capacity event should not be created if the `nostop` is not equal 0[8];
- If the Scheduling Module is "coupled" with a Resource Handling Module, this function must update the ceiling (for example as done with the SRP protocol).

```
void (*public_epilogue)(LEVEL l, PID p);
```

This function is called by the Generic Kernel when:

- the running task p is preempted by another task in the system;
- a function that may generate a preemption is called (these functions usually call the generic scheduler that, as a first operation, simply call this function);
- the capacity of the running task p is exhausted (the capacity exhaustion is handled as a preemption request!).

In general, this function receives as an additional parameter the running task index. The effect of the call may also disable the schedulability of the task. When this function is called the task p has a state equal to EXE.

The typical actions done by this function are:

- if the schedulability of the task is still active, the task is reinserted into the ready queue, if there exists one into the Module;
- the state of the task is modified and it is set to an internal Module state (for example, a READY state);

[8]In this case the task is going to be the running task because it locks a resource needed by a high priority task, so usually the task must execute until the release of the blocking resource.

- the capacity of the task is checked: if it is exhausted some operations will be done, for example an exception is raised, a deadline is postponed, and so on;
- If the Module created a capacity event without using the `CONTROL_CAP` field and without using the `cap_timer` variable, that capacity event must be removed.

```
void (*public_activate)(LEVEL l, PID p, struct timespec *t);
```

This function is called by the Generic Kernel when an explicit activation for the task is called using the `task_activate` primitive or the `group_activate` primitive.

The PID parameter is the task that has to be activated. Also, the activation time is given as an parameter (this will be the time-base of the task). The effect of the function is to activate the schedulability of the task.

The typical actions done by this function are listed below:

- First, a check is done to verify if the task is in a state compatible with the activation (for example, a sporadic task cannot be activated too frequently);
- The state of the task, usually equal to SLEEP, is modified (for example it becomes READY);
- If the Module has a ready queue, the task is inserted in it;
- If the Module counts the task's pending activations, and the task does not have finished his current activation yet, the activation should be saved for the task;
- If the Module handles periodic tasks or tasks with temporal deadlines, some events should be created to check these conditions.

```
void (*public_block)(LEVEL l, PID p);
```

The function implements the blocking of the task p (p is the running task) in a generic synchronization primitive that does not use the shadow mechanism.

The function must disable the schedulability for the task until it is "freed" by a call to the `public_unblock` function.

The typical actions done by this function are:

- The task should be extracted from the ready queue if the `task_dispatch` didn't do this;
- The events posted with the dispatch should be removed (for example, the capacity events should be removed, but not the deadline ones);
- The function shall not modify the state of the task; the state of the task is modified by the primitive that calls the function;

```
void (*public_unblock)(LEVEL l, PID p);
```

The function accepts a parameter p that is the task that has terminated the synchronization started with a call to the `public_block` function. After this call, the awaken task can be scheduled in the system.

Usually the function inserts the task into some internal queues, and the state of the task is modified (for example it is set to READY). Usually the function does not post any event.

The function is called into the code of the Generic Kernel primitives that implement synchronization without using the shadow mechanism. The Generic Kernel guarantees that this function is not called before the corresponding call to `public_block`.

```
int (*public_message)(LEVEL l, PID p, void *m);
```

This function is called when the task_message() primitive is called by a task to send a message to the scheduler. Typical messages are, for example, the end of an instance, or some kind of signaling the task must do to a scheduling module, like a checkpint mechanism.

The parameter p is the running task (that is, exec_shadow). A parameter m is passed, and it can be used to pass arbitrary parameters to the scheduler. The value NULL is typically used by the Kernel to signal the task_endcycle primitive. An integer is also returned to return a kind of status value to the calling task.

The task_endcycle primitive

A typical message that a task sends to a scheduling module is the end of an instance, signaled using a task_endcycle() primitive. The implementation of that primitive is simply a "task_message(NULL,1)". The task_message should implement a behavior similar to the one descried in the following paragraphs.

If the task does not have pending activations (or if the Module does not manage them) the effect of the function is to disable the schedulability of the task until an explicit activation is done by the user[9] or an automatic reactivation done by the Module (if the task is periodic).

If the task has some pending activations the function will reactivate the task in a way similar to `public_epilogue`.

The typical actions done by this function are listed below:

- If the task must be suspended, the task state is modified to a "parking" state (`IDLE` or `SLEEP`);
- The task may be removed from the ready queue (if it was not removed by the `public_dispatch`);
- Some resource reclaiming algorithm may be implemented, because a task instance is finished;
- Timer events related to the budget exhaustion and deadlines are handled (for example events can be deleted, or postponed).

4.3.2 Private Functions

This section describes the private functions provided by a scheduling module. Private functions are called only by other scheduling modules, and never by the generic kernel, and they represent the interface extorted by a scheduling module towards other modules. Typical example of use of the private functions are:

- an EDF module have to implement task activation and unblocking simply inserting a task into the ready queue. For that purpose, the functions public_activate and public_unblock internally calls the private function private_insert. Moreover, the two functions can have some peculiarities. For example, the implementer would like that public_activate adds a deadline check posting an OSLib event at the deadline time, whereas this is not the case of public_unblock, because the deadline used before have to be used again. Deadline posting can be done into public_activate. private_insert will simply insert the task into its "private" queue.

[9] using a `task_activate` call.

- an aperiodic server wants to insert a task into the EDF queue. Again, the public_activate and the public_unblock of the aperiodic server will call the private_insert of the EDF queue, that will insert the task into its queue.

```
void (*private_insert )(LEVEL l, PID p, TASK_MODEL *m);
```

This function is used to insert a task into the Module. The inserted task must have been already created through a call to the `task_create` primitive. When inserted, the behavior of the function is as the task has been activated into the module (e.g., the task goes into the ready queue). All the useful informations passed to the task Model must be registered internally to the Module.

```
void (*private_extract )(LEVEL l, PID p);
```

This function terminates a chunk of a task previously inserted in the Module using private_insert. The typical effect of this function is to extract the task from the internal queues and to delete the events generated for the task (deadline, capacity, and so on). The function shall not insert the task descriptor in the `freedesc` queue[10].

```
void (*private_dispatch)(LEVEL l, PID p, int nostop);
```

This function is usually called by `public_dispatch` to inform the Master Module that a task inserted in it as a guest is being dispatched. The semantic of the function is similar to that of `public_dispatch`.

```
void (*private_epilogue)(LEVEL l, PID p);
```

This function is called by a Module to inform the Master Module that a task inserted in it using private_insert() is being preempted or its budget has been exhausted.

```
int (*private_eligible)(LEVEL l, PID p);
```

This function is usually called by public_eligible to inform the Master Module that a task inserted in it as a guest has been chosen for scheduling. The semantic of the function is similar to that of `public_eligible`.

4.4 Registration Function

The code contained in a Module is composed by the function calls (Level, Task and Guest Calls), and by a Registration Function that must be called when the system starts to properly initialize the kernel data structures. This registration function can be thought as a C++ constructor for the Module.

A Module initialization typically consists of four parts:

- The first part allocates a level descriptor that will be used to initialize the Module; To alloc a level descriptor, the functions
  `LEVEL level_alloc_descriptor(void)` and
  `RLEVEL resource_alloc_descriptor(void)`
  must be used. These functions take no arguments and return a free descriptor to be used;

[10] public_end() is responsible for that...

- The second part initializes the function pointer of the level descriptor of a scheduling module;
- The third part initializes the private data structures of the Module, and possibly posts a function that has to be called just after the system has gone in multitasking mode;
- The fourth part executes the function posted in the third part.

The first three parts are written in the Registration Function, which is called by the `__kernel_init__` function before the Kernel goes in multitasking mode.

In the `__kernel_init__` function no Generic Kernel primitives can be called. If there is a need to do that, a registered function has to be called instead. For example, the dummy task is created by the Dummy Module (`kernel/modules/dummy.c`) through a registration function. The main task is created in the same way by the Round Robin Module (`kernel/modules/rr.c`).

4.4.1 Default values

In general, the scheduling modules needs only to register the functions they want to redefine. For that reason, the public and private functions have a default value, set by `level_alloc_descriptor()`. Here is a list of these default values:

private_insert Kernel Exception.

private_extract Kernel Exception.

private_eligible Returns 0 (that is, the task can be accepted for scheduling).

private_dispatch Kernel Exception.

private_epilogue Kernel Exception.

public_scheduler Returns -1 (that is, no task are ready to be scheduled).

public_guarantee Returns 1 (that is, the system can be scheduled).

public_create Returns -1 (that is, the model can not be handled by the module).

public_detach Does nothing.

public_end Kernel Exception.

public_eligible Returns 0 (that is, the task can be accepted for scheduling).

public_dispatch Kernel Exception.

public_epilogue Kernel Exception.

public_activate Kernel Exception.

public_unblock Kernel Exception.

public_block Kernel Exception.

public_message Kernel Exception.

4.5 Writing Conventions

This Section explains some conventions followed in writing the Modules. They are useful to understand how to write new Modules using the same style adopted in the Modules distributed with the Kernel. They can be summarized as follows.

- Each Module is composed of two files, one file `.h` and one file `.c`. The `.h` files are stored in the `include/modules` directory; the `.c` files are stored in the `kernel/modules` directory.
- Each Registration Function registers only ONE level descriptor. In this way the level at which a Module is registered can be found inspecting the initialization file. The level descriptor number is usually returned by the register function.
- The Task Models used by the Modules are listed in the `include/kernel/model.h` file.
- The names of the internal functions are defined as `static` and are in the form `MODULENAME_FUNCTIONNAME`, where `MODULENAME` is the name of the `.c` file where the Module code is written.
- A Module can export some functions to implement a specific behavior; these functions have a first parameter of type `LEVEL`, in order to retrieve the Module data structures. An application that relies on a specific Module configuration can use the functions under the assumption that the Application knows the level at which the Module is registered.
- The prototypes of the functions exported by a Module (registration function plus other functions if present) have to be included in the `.h` file.
- The Modules should not use global data, because different instances of a Module can be registered at the same time in the system.

In general writing a new Scheduling Module requires the creation of new files. To simplify the distribution and the integration of new modules in the Generic Kernel no modifications have to be made to the standard distribution of the Kernel. Beside that, a few rules of common sense have to be followed:

- A new Scheduling Module should consist of only one `.h` file and only one `.c` file. Modules composed of many files should contain an explanation in their documentation.
- Together with the Scheduling Module a system designer should provide:
  - an initialization file (similar to those present in the `kernel/init` directory) that shows how the new Module must be initialized;
  - at least one test program showing the functionality of the Module;
- New data definitions (for example new Task Models, new `pclass`, version and exception values) used by the new Modules should be inserted into the `.h` file of the Module. New constants should be different from the others contained into the default distribution.

A template example of a Scheduling Module can be found on the S.Ha.R.K. web site. Examples of third-party scheduling modules can be dound into the demos directory (e.g., demos/static, demos/edfact, demos/cash).

Chapter 5

Resource Modules

In this chapter the interface of a Resource Module is described. The semanthic of the various functions, and the approach used to handle the Shared Resource Access Protocols are described. For architectural informations look at Sections?? and ??.

5.1 Resource Modules Interface

The approach used to define the interface of a Resource Module is similar to that used for the Scheduling Modules (look at Section 4, 4.3 and 4.5).

The interface of a Scheduling Module is less complex than that of the Scheduling Modules: only the *create* and *end* events are handled in the task life. This choice is made because the Resource Modules usually does not influence the scheduling of the system[1].

All the functions of a Resource Module are called with Interrupt disabled.

```
int (*res_register)(RLEVEL l, PID p, RES_MODEL *r);
```

This function is called by the Generic Kernel by the `task_create` primitive to register a Resorce Model into a Resource Module.

The function will accept as parameter the index p of the descriptor allocated by the Generic Kernel for the task and one of the Resource Models passed through the primitive.

The Generic Kernel guarantees that the Resource Model passed in this function can be handled by the Module. The function returns 0 if the Module can handle the request, -1 if the task can not be created because the Module can not guarantee the quality of service required.

The function will set up the local Module data with the parameters of Quality of Service passed with the Resource Model.

```
void (*res_detach)(RLEVEL l, PID p);
```

The call of this function signals to the Module that the task p is terminated, so all internal data structures must be updated.

This function is called independently from the fact that the task has or not registered some Resource Model in the Module in two cases:

[1] Note that a Resource Handling Algorithm that modifies the scheduling of the system is more like a Scheduling Module than a Resource Module. Some hybrid approaches can be implemented requiring that the implementation is made using two Modules that can modify the private data of each other.

- The primitive `task_create` fails for some trouble inependent from the Module; in this case the function is called before the `public_detach` function of the Scheduling Module that owns the task;
- The task terminates in the correct way; in this case the function is called before the function `public_end` of the Scheduling Module that owns the task.

In other words, this function implemets the behaviour of the Scheduling Module's `public_detach` and `public_end` functions. This is correct because the Resource Module only react to the creation and termination events. It doesn't matter if the task is terminated correctly or if it has not been created...

5.2 Implementation of the Shared Resource Access Protocols

The interface exported by the Resource Modules is used also by the Modules that implements the Shared Resource access Protocols.

The problem solved developing these Modules is the project of some OS primitives that can be independent from the used protocol, and, moreover, independent from a specific Module registered at run-time.

5.2.1 Used Approach

The approach used is to extend the Resource Module interface; in this way also the protocols that requires some per-task parameters can be implemented[2].

Using an Object Oriented approach the hierarchy of the Modules can be described (look at Figure5.1).

These Modules are viewed by the Generic Kernel as Resource Modules. When a mutex is initialized some checks[3] are done to find the Module that extends the interface in the correct way[4]and therefore implement the required protocol.

5.2.2 The mutexes

The mutexes are stored in the `mutex_t` structure showed in Figure 5.2. That declaration is contained in the file `include/kernel/descr.h`. The structure contains the following fields:

mutexlevel It is the level which the Module is registered in.

use It tells if the mutex is currently used into a synchronization done through condition variables.

opt This field is a pointer to a structure that the Module can dinamically alloc to handle protocol-dependent parameters[5].

[2]For example, these parameters can be the ceiling of a task on a Priority Ceiling or SRP protocol...

[3]Similar to those used in the primitive `public_create` for the Task Models.

[4]To know that a Module has extended the Resource Modules Interface the `rtype` field is provided (look at Section 3.2.3).

[5]These parameters cannot be allocated internally to the Module because the mutexes are not statically allocated as task or semaphore descriptors.

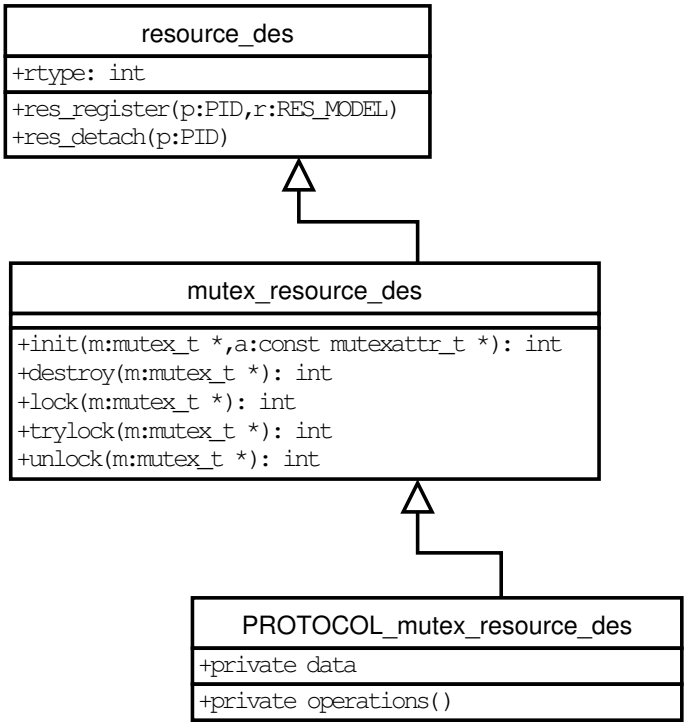

Figure 5.1: UML Diagram that shows the class hierarchy that implements the Shared Resource AccesProtocols.

```
typedef struct {
   RLEVEL mutexlevel;
   int use;
   void *opt;
} mutex_t;
```

Figure 5.2: The `mutex_t` structure.

5.2.3 Interface extension

In this section the extension to the Resource Modules is described. The proposed functions handles all the events that belongs to amutex.

Only the function `init` is called with interrupts disabled. The other functions have to disable the interrupts, because there is not a generic behaviour for all these functions (for example, the lock of a mutex may be non-blocking on some protocols).

```
int init(RLEVEL l, mutex_t *m, mutexattr_t *a);
```

This function is called to init a mutex with a protocol. The function accepts as parameters the mutex to be initialized and the mutex attribute that store the parameters to be used in the initialization.

The function returns a value of `0` if the mutex initialization was successful, or an error code otherwise. The error codes returned by the functions must be compatibles with the functions `pthread_mutex_init` of the POSIX standard.

```
int destroy(RLEVEL l, mutex_t *m);

int lock(RLEVEL l, mutex_t *m);

int trylock(RLEVEL l, mutex_t *m);

int unlock(RLEVEL l, mutex_t *m);
```

These functions implements the core functionality of the mutexes and they have a semantic similar to the corresponding POSIX functions. In particular they receive as parameter a pointer to a mutex, and they returns 0 if the operation was successful or an error code if not.

These functions have to manage internally the context change in the system, because it is not possible to give a fixed rule for these functions (for example, the lock operation, that usually can block the task, is never a blocking primitive under the SRP assumptions).

Chapter 6

Examples

The application of the proposed approach is presented in this chapter, on three meaningful examples, by showing the code of the scheduling and of the resource modules. The examples are really implemented in the Kernel, so these remarks can also be used as documentation.

6.1 SIMPLEEDF (Earliest Deadline First) Scheduling Module

This section describes an implementation of EDF with the following characteristics:

- Support for periodic and sporadic tasks;
- Support for release offsets and relative deadlines less than the period;
- On-line guarantee using the utilization factor paradigm;
- Temporal isolation implemented using the `CONTROL_CAP` flag;
- A number of different deadline/WCET/activation violation options.

Typically this module is registered at Level 0; in effect the guarantee algorithm works only if the Module can use all the bandwidth of the system. In order to schedule background periodic tasks, a soft task model should be used with a Scheduling Module like the CBS Scheduling Module. The Module described in this section is contained in `modules/edf.c`.

6.1.1 State transition diagram

The state transition diagram for an EDF task is shown in Figure 6.1.

The states whose name start with `EDF_` are internal Module statuses. The event names are reported near the arcs; in particular the names of the timer events are written within parentheses. The different states have the following meanings:

`FREE` Before creation, and after destruction, the task is in this state.

`SLEEP` The task waits in this state for an explicit activation.

`EDF_READY` This is the classic ready state where the task has to wait until it has the highest priority (i.e., the earliest deadline).

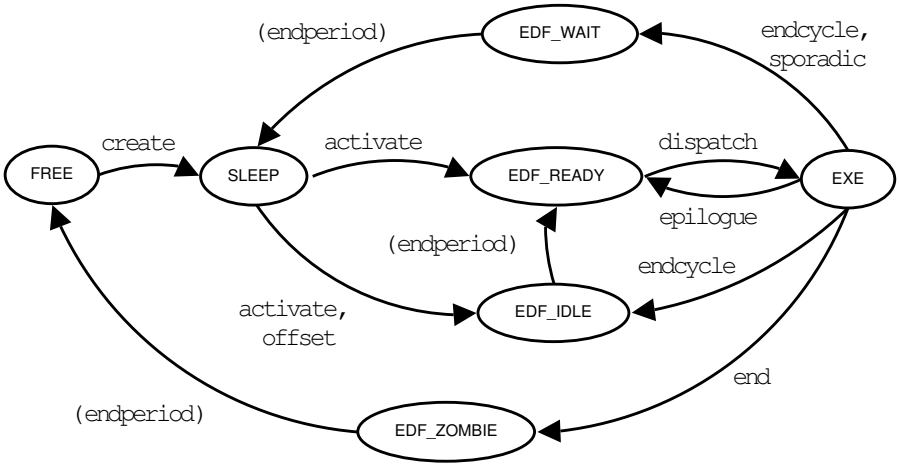

Figure 6.1: State transition diagram for an EDF task. (For simplicity, some transitions have been excluded.)

`EXE` This is the state when the task is executing.

`EDF_WAIT` This is the state of a sporadic task after finishing an instance, waiting for the endperiod event to arrive.

`EDF_IDLE` This is the state of a periodic task after finishing an instance, waiting for the endperiod event to arrive. An activated task with a release offset is also put in this state, waiting for the first period to arrive.

`EDF_ZOMBIE` This is the state where a task is put when it terminates correctly. The allocated bandwidth is freed when the endperiod event is fired.

6.1.2 Level descriptor

The EDF Module extends the `level_des` structure that contains the interface exported by a generic Scheduling Module. The extended data structure, `EDF_level_des`, contains the following fields:

`flags` This variable stores the flags passed to the module at registration time. The following flags are defined:

- `EDF_ENABLE_DL_CHECK` If set, the module will keep track of task deadlines by internal deadline timers. The behavior in case of a deadline overrun depends on the `EDF_ENABLE_DL_EXCEPTION` flag, see below.
- `EDF_ENABLE_WCET_CHECK` If set, the module will keep track of task execution times by enabling the CONTROL_CAP flag for the tasks in the generic kernel.
- `EDF_ENABLE_DL_EXCEPTION` If set, the module will raise an exception if a deadline overrun occurs. If not set, the `dl_miss` counter for the task is increased every time a deadline overrun occurs.

`EDF_ENABLE_WCET_EXCEPTION` If set, the module will raise an exception if an execution-time overrun occurs. If not set, the `wcet_miss` counter for the task is increased every time an execution-time overrun occurs.

`EDF_ENABLE_ACT_EXCEPTION` If set, the module will raise an exception if a task is activated more often than its declared minimum interarrival time. If not set, the `nskip` counter for the task is increased instead.

`ready` This is an `IQUEUE` variable used to handle the ready queue.

`U` This variable is used to store the sum of the reserved bandwidth for the tasks owned by the Module.

`tvec` A vector of EDF task descriptors, `EDF_task_des`, that defines a number of additional variables for each task, see below.

6.1.3 Task descriptor

The EDF module introduces a number of additional task variables. They are collected in a `EDF_task_des` structure for each task:

`flags` Flags that store some additional type/status information about the task. The following flags are defined:

`EDF_FLAG_SPORADIC` The task is sporadic. This influences some state transitions, see Figure 6.1.

`EDF_FLAG_SPOR_LATE` The task is sporadic and has experienced a period overrun. (This is only possible if the EDF_ENABLE_DL_EXCEPTION level flag is not set.) When finished, the task should go directly to the SLEEP state.

`period` The period or minimum interarrival time of the task.

`rdeadline` The relative deadline of the task. Currently, only $D \leq T$ is allowed.

`offset` The release offset, relative to the activation time of the task.

`release` This variable stores the release time of the current instance.

`adeadline` This variable stores the absolute deadline associated with the most recent task activation.

`dltimer` A handle to the task deadline timer.

`eop_timer` A handle to the task end-of-period timer.

`dl_miss` Counter for the number of missed deadlines.

`wcet_miss` Counter for the number of execution-time overruns.

`act_miss` Counter for the number of skipped activations.

`nact` The current number of queued activations (periodic tasks only).

6.1.4 Module internal event handlers

The module uses internal kernel events (one-shot timers) to handle the release of tasks, deadline overruns, etc. When an event is posted by the module, an event handler is specified, and the PID of the relevant task is passed as parameter. For instance, the `endperiod` event is handled by the following function:

```
static void EDF_timer_endperiod(void *par)
```

The first thing to do when handling an event is to recover the information about the Module that posted the event. This can be done with the following statements:

```
PID p = (PID) par;
EDF_level_des *lev =

          (EDF_level_des *)level_table[proc_table[p].task_level];
```

The generic kernel task fields can be referenced as `proc_table[p].name`; the internal data of the Module can be referenced as `lev->field`. The EDF task descriptor can be referenced as

```
EDF_task_des *td = &lev->tvec[p];
```

and the EDF task fields can then be referenced as `td->field`.

By studying the state transition diagram, we see that different actions should be taken depending on the state of the task:

- If the task state is `EDF_ZOMBIE`, the task is inserted into the `freedesc` queue and the allocated bandwidth is freed;
- If the state is `EDF_WAIT`, the task state is set to `SLEEP`, so the sporadic task can be reactivated;
- If the state is `EDF_IDLE` and the task is periodic, it is reactivated by a call to the `EDF_intern_release` function. This involves posting a deadline event (if `EDF_ENABLE_DL_CHECK` is set); inserting the task in the ready queue; increasing the absolute deadline; and telling the kernel that it may need to reschedule by calling `event_need_reschedule`.
- If the state is `EDF_IDLE` and the task is sporadic, it is marked as late.

The module also contains the following event handlers:
static void EDF_timer_deadline(void *par)
static void EDF_timer_offset(void *par)
static void EDF_timer_guest_deadline(void *par)

6.1.5 Public Functions

The Module redefines the Level Calls interface. In the following paragraphs the implementation of these functions is described.

All the functions of the interface receive a parameter of type `LEVEL` that can be used in a way similar to the parameter passed to the event functions to find all the data structures needed.

```
PID EDF_public_scheduler(LEVEL l);
```

This is the Module scheduler that, as all good schedulers, simply returns the first task in the ready queue without extracting it.

```
int EDF_public_guarantee(LEVEL l, bandwidth_t *freebandwidth);
```

The on-line guarantee function simply verifies if there is enough free bandwidth for scheduling its tasks. If so, the free bandwidth is decremented by the amount used by the Module, and it returns that the task set can be guaranteed.

If the guarantee is called after a task creation in the Module, it can be the case that the new task, with all the other tasks already guaranteed by the Module, uses a bandwidth greater than 1 (note that the U field can store only numbers in the $[0 \ldots 1]$ interval). In this case, in the function `EDF_public_create` a flag is set forcing the guarantee algorithm to fail.

```
int EDF_public_create(LEVEL l, PID p, TASK_MODEL *m);
```

The function checks if the Model passed as second parameter can be handled. In this case, the Module handles all the HARD_TASK_MODELs that have a correct `pclass` value and a wcet and a period != 0. This function sets the `period`, `flag`, and `wcet` internal fields of the newly created task. The function sets also the `CONTROL_CAP` flag to inform the Generic Kernel that the execution time have to be controlled. Finally, the function allocates the system bandwidth in such a way that it can be checked by the guarantee algorithm. If the bandwidth allocated for the already guaranteed tasks plus the new one is greater than 1, a flag is set to signal that the guarantee algorithm must fail.

```
void EDF_public_detach(LEVEL l, PID p);
```

This function simply reclaims the bandwidth used by the task allocated by `EDF_public_create`, disabling the flag set by `EDF_public_create` when the guarantee is impossible (U>1).

```
int EDF_public_eligible(LEVEL l, PID p);
```

This function simply returns 0 because the EDF tasks are always eligibles. In fact, the EDF Module does not use the guest functions of another Module to handle its tasks.

```
void EDF_public_dispatch(LEVEL l, PID p, int nostop);
```

To dispatch an EDF task, the task itself must be removed from the ready queue. The capacity handling (like the capacity event post) is automatically done by the Generic Kernel.

```
void EDF_public_epilogue(LEVEL l, PID p);
```

The function must suspend the running task because it has been preempted or because if has finished his capacity. Therefore, the first thing to be done is the check of the available capacity. If it is exhausted an exception is raised and the task is put into the `EDF_WCET_VIOLATED`[1] state.

When the task has not consumed all of its capacity it is inserted back into the ready queue.

[1]The task shall not be extracted by any queue because the task was extracted by the `EDF_public_dispatch` function.

```
void EDF_public_activate(LEVEL l, PID p, struct timespec *t);
```

This function simply activates the task, inserting it into the ready queue. A task can be activated only if it is in the `SLEEP` or in the `EDF_WCET_VIOLATED` state. If the task is in the `EDF_WAIT` state it means that the task is a sporadic task activated too early, so an exception is raised.

The function executes the following steps:

- A suitable deadline is computed for the task;
- The task is inserted into the ready queue;
- A deadline event is posted for the task.

```
void EDF_public_unblock(LEVEL l, PID p);
```

The function simply inserts the task into into the ready queue. The task was blocked on a synchronization by a call to the function `EDF_public_block`.

```
void EDF_public_block(LEVEL l, PID p);
```

The function implements a synchronization block. The function simply does nothing, because:

- The task was already extracted from the ready queue;
- The capacity event is handled by the Generic Kernel;
- The task state is set by the calling primitive;
- The deadline does not need modifications.

```
int EDF_public_message(LEVEL l, PID p, void *m);
```

This function implement only the task_endcycle behavior, doing two things:

- The `wcet` of the task is refilled, since the task has finished its instance;
- The task state is set to `EDF_IDLE` or `EDF_WAIT`, waiting the deadline arrival.

```
void EDF_public_end(LEVEL l, PID p);
```

The function should erase all the information about the task in the data structure of the Module. It simply sets the task state to `EDF_ZOMBIE`. The deallocation of the bandwidth used by the task and the freeing of the task descriptor is performed while handling the deadline event.

6.1.6 Private Functions

The EDF Module can accept a JOB_TASK_MODEL as the Model for the Guest Tasks. This Model does not provide information about the time that the task will execute. This means that the EDF Module does not check the execution time of the task. It must be checked by the Module that inserts the tasks as guest tasks. In the following paragraphs the guest calls are described.

```
int EDF_private_insert(LEVEL l, PID p, TASK_MODEL *m);
```

This function is called by a generic Aperiodic Server Module to insert a task into the EDF Module.

The function simply fills the private data structures of the EDF Module with the task parameters passed through the Task Model. No guarantee is done on the guest tasks (the guarantee of a guest task is a responsibility of the Module that calls this function).

```
void EDF_private_dispatch(LEVEL l, PID p, int nostop);
```

This function is typically called by the `task_dispatch` Task Call of the Module that inserts the task as guest task. The effect of the call is to extract a task from the ready queue, so it is identical to the `task_dispatch` Task Call.

```
void EDF_private_epilogue(LEVEL l, PID p);
```

This function is called when a task is preempted (no capacity are handled by the Module). The function simply inserts the task into the ready queue.

```
void EDF_guest_activate(LEVEL l, PID p);
```

This function is called when the Module that inserts the task in the system through the EDF_private_insert wants to activate it. The effect of this function is to insert the task into the ready queue and to post a deadline event if the deadline miss should be detected for the task (a flag that specifies if the Module should generate a deadline is given in the JOB_TASK_MODEL.

```
void EDF_private_extract(LEVEL l, PID p);
```

This function is called by a Module when it wants to terminate a task inserted as guest. This function is not called only at task termination, but also when the Module has to change some parameters for the task (for example, when a deadline should be postponed).

The function has to erase all references to the task in the private data structures of the EDF Module. Note that this function can be called also when the task is not in the EXE state, so all task states should be checked for a correct behavior.

6.1.7 Additional Functions exported by the Module

Finally, the EDF Module exports two functions that can be used by the user.

The first function is the registration function, used to register the Module into the system and to init the internal data structures.

The second function, `EDF_usedbandwidth`, can be used to get the bandwidth actually used by the Module. That function needs as a parameter the level at which an EDF Module is registered, so all the private data structures can be read. Note that giving the level of a Module in an application should not be considered a violation of the independence of an Application to the Registered Modules. If an application wants to know a specific data in a Module, it has to know in what level the Module is Registered...

6.2 PS (Polling Server) Scheduling Module

In this Section will be described an implementation of a PS Module that have the following characteristics:

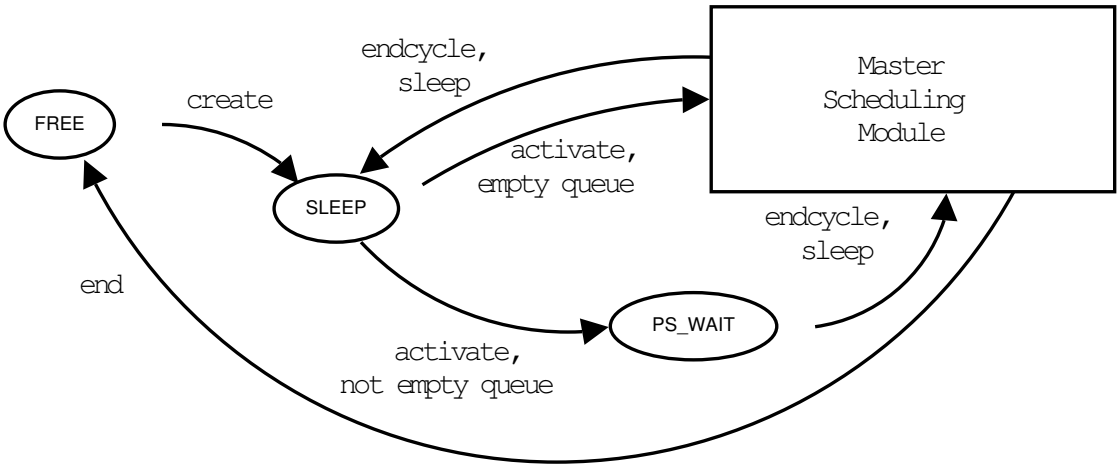

Figure 6.2: State Diagram of a Task scheduled with the PS Module.

- Soft Aperiodic Task support;
- On-line guarantee using the utilization factor paradigm;
- Temporal isolation implemented without the `CONTROL_CAP` flag.
- Feature that allows to use the idle time left by the Modules registered at lower level numbers;
- Support for pending task activations;
- Compatibility with static (RM) and dynamic (EDF) algorithms.

This Module implements an aperiodic server that inserts its tasks into another Scheduling Module, without having any information on how the Master Module is implemented. The module described in this section is contained in `modules/ps`.

6.2.1 Transition state diagram

In Figure 6.2 the state diagram of a task is showed.

The Module introduce only one state, `PS_WAIT`. This is the state in which a task waits to be inserted in the "ready queue". The server queue is handled in a FIFO way. The other states in which a Module will go are internal states of the Master Module.

6.2.2 Private Data structures

The PS Module redefines the `level_des` structures adding some private data structures, listed below:

`nact` This array is used to track the pending activations of a task handled by the Module. Each task element has a value of `-1` (if the task skips the pending activations) or a value `>=0` (that is, the value of pending activations for the task);

`lastdline` This field is used to store the deadline used by the Polling Server;

`period` This field stores the reactivation period of the server;

`Cs` This field stores the Maximum capacity of the server;

`availCs` This field stores the computation time available for the server at a given time. The capacity is updated when the task handled by the Module is scheduled by the Master Module, or when a task handled by the Module is dispatched following the shadow chain and the server is not scheduling in background;

`wait` This field is the wait queue, where the tasks wait their turn to be served by the Polling Server;

`activated` This field is the PID of the task currently inserted into the Scheduling Module. The server inserts in the Master Module maximum one Module at a time;

`flags` The flag field is used to store some informations passed when the Module was registered, like for example the implemented guarantee algorithm (RM or EDF). This field is used also to know if the server is executing a task in background;

`U` This field is used to store an information of the used bandwidth by the task handled by the Module.

`scheduling_level` This field stores the level of the Host Module.

6.2.3 Internal Module Functions

The PS Module needs to define some internal functions. These functions are called to handle the internal events posted by the Module. Many of these functions are declared `static`, so they are not visible externally to the Module.

```
void PS_activation(PS_level_des *lev);
```

This function is an inline function and it handles the activation of a task into a Master Module. In particular:

- It inits a Job Task Model that will be passed to the Master Module with the period and the deadline of the Polling Server;
- It creates and activates through the guest calls the task indexed by the private field `lev->activated`.

```
void PS_deadline_timer(void *a);
```

This function implements the periodic reactivation of the Polling Server. In particular:

- a new deadline is computed and a new deadline event is posted;
- the Server capacity is reloaded to the maximum value if the available capacity was positive, to a value less than the maximum (a "recharge" (sum) is done) if negative;
- If the recharge turn the available capacity positive, a waiting task is activated, or the capacity available is depleted.

6.2.4 Public Functions

All the functions of the interface receives a LEVEL parameter that can be used in a way similar to the parameter passed to the event functions to get all the private data of the Module.

```
PID PS_level_scheduler(LEVEL l);
```

This scheduler is used by the Module if the Module is registered specifying that it can not use the idle time left by other Modules. It always return `NIL` (meaning that the module has nothing to schedule).

```
PID PS_level_schedulerbackground(LEVEL l);
```

This scheduler is used by the Module if the Module is registered specifying that it can use the idle time left by other Modules. It sets a flag in the `flags` field to remember that the Module is scheduling a task in background, after that it returns the first task in the wait queue. The scheduler is disactived in case the task scheduled by the server is blocked on a synchronization (look at the function `PS_public_block`).

```
int PS_level_guaranteeRM(LEVEL l, bandwidth_t *freebandwidth);
```

```
int PS_level_guaranteeEDF(LEVEL l, bandwidth_t *freebandwidth);
```

These two functions implements the internal guarantee of the Module, simply decrementing when possible the available bandwidth. the difference between the two functions is that the first function allow to schedule until a used bandwidth of 0.69, and the second one allow a scheduling limit of 1. The structure of this function is similar to that of `EDF_public_guarantee`, except that the guarantee is done on the server and not on each task handled by the server..

6.2.5 Task Calls

```
int PS_public_create(LEVEL l, PID p, TASK_MODEL *m);
```

This functions checks if the Task Model passed can be handled by the Module. The Module simply handles all the Soft Task Models that specify a periodicity equal to APERIODIC, ignoring each additional parameters. This function sets the `nact` field on the new task according to the requirements of the task model. The function does not set the flag `CONTROL_CAP`.

```
void PS_public_detach(LEVEL l, PID p);
```

This function does nothing, because in the `PS_public_create` function no dynamic data structures were allocated, and because the tasks served by do not require individual guarantee (the guarantee is done for the whole Server).

```
int PS_public_eligible(LEVEL l, PID p);
```

The function always returns 0 because the PS tasks are always eligible.

```
void PS_public_dispatch(LEVEL l, PID p, int nostop);
```

A task can be dispatched if it is in the wait queue or if it is inserted into the Master Module.

In the first case the task is extracted from the `wait` queue, in the latter case the private function `private_dispatch` is called to signal to the Master Module to dispatch the task.

Then, a capacity event is generated (only if the parameter `nostop` is `0` (the parameter is 0 if there is no substitution due to the shadow chain mechanism). The capacity event is created using the `cap_timer` field. In this way the event will be removed by the Generic Kernel.

```
void PS_public_epilogue(LEVEL l, PID p);
```

This function implements the suspension of the running task due to a preemption or a capacity exhaustion.

First, the function updates the server capacity using the `cap_lasttime` and `schedule_time` variables. If the server capacity is exhausted, the task that is inserted into the Master Module (if the task is not scheduled in background) terminates with a `private_extract`, and then it is inserted in the wait queue.

If the server capacity is not exhausted, there are two cases: if the Module is scheduling a task in background, the task will be inserted on the head of the wait queue, otherwise the `private_epilogue` function will be called to signal the Master Module a task preemption.

```
void PS_public_activate(LEVEL l, PID p);
```

This function activates a task. If the task passed as parameter is the task currently inserted in the Master Module, or if the task is already inserted into the `wait` queue, the activation is stored and the nact field is incremented (only if the task has specified in the task model passed at its creation to save the activations).

Otherwise the normal activation actions are done:

- the field `request_time` of the task descriptor is updated;
- The task is activated using the internal `PS_activation` function if there aren't tasks in the Module and the server capacity is positive, otherwise the task is queued into the wait queue.

```
void PS_public_block(LEVEL l, PID p);
```

The function should implement the synchronization blocking. The function should block the whole server, calling eventually the `private_extract` guest call on the Master Module and setting the `PS_BACKGROUND_BLOCK` flag to block every background schedule.

```
void PS_public_unblock(LEVEL l, PID p);
```

The function should reactivate a blocked task. The task reactivation consists of its insertion into the `wait` queue and of the reset of the flags set in the `public_block` function.

```
int PS_public_endcycle(LEVEL l, PID p, void *m);
```

The function implement the task_endcycle behavior and does the following steps:

- The server capacity is updated, or the background scheduling feature is disabled if it was active;

- The task was extracted from the Master Module through a call to the private function `private_extract`, otherwise it is extracted from the `wait` queue;
- If the task has some pending activations, it is inserted at the end of the `wait` queue, otherwise the task state is set to `SLEEP`.
- If possible a new task is extracted from the top of the `wait` queue, through a call to the function `PS_activation`;

```
void PS_public_end(LEVEL l, PID p);
```

The function directly insert the task into the `freedesc` queue.

6.2.6 Private functions

The private functions are not defined (the defaults are used).

6.2.7 Functions exported by the Module

Finally, the PS Module exports two functions that can be used by the user.

The first function is the registration function, used to register the Module in the system and to init the internal data structures. This function registers also a initialization function that posts the first server deadline event. This event cannot be created in the registration function because the registration function is called when the OS Lib is not initialized yet.

The second function, `PS_usedbandwidth`, can be used to obtain the allocated bandwidth of the Module, and it is similar to the function `EDF_usedbandwidth`.

6.3 PI (Priority Inheritance) Resource Module

In this Section an implementation of the shared resource access protocol Priority Inheritance (PI) [8] is described; it has the following characteristics:

- support for the Generic Kernel mutex interface;
- use of the shadow mechanism provided by the Generic Kernel;
- independence from the data structures used internally by the Scheduling Modules;
- possibility of a static initialization of the used mutexes.

The Module is contained in `modules/pi`.

6.3.1 Used approach

The key idea of the implementation are:

- when a task enters into a critical section locking a mutex, the Module registers which is the task that owns the mutex, because it is used if other tasks try to lock the mutex, and because only that task can unlock it;
- the Module registers the number of mutexes that owns the tasks, to check that the task does not die with some mutexes locked.

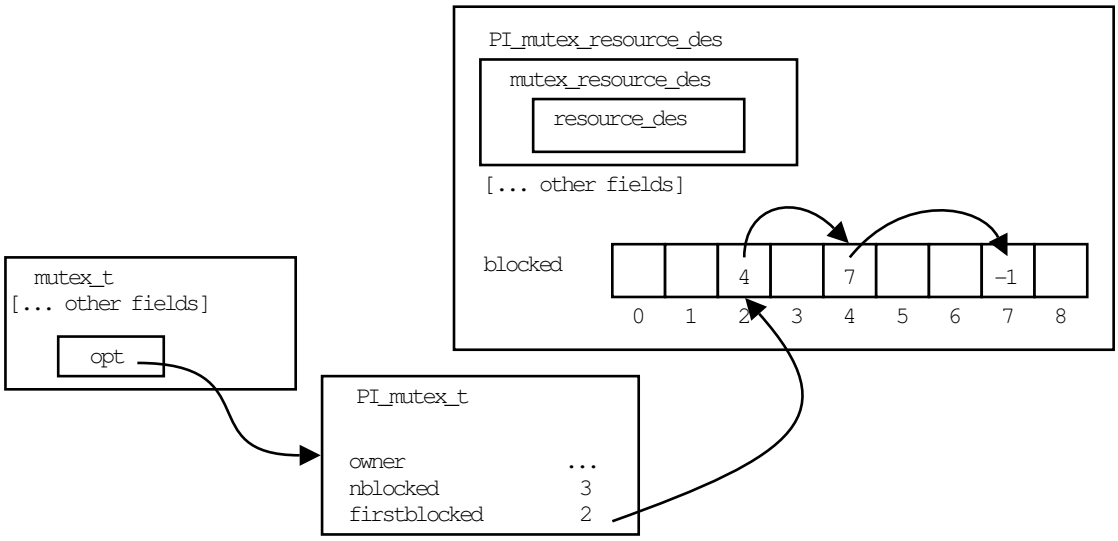

Figure 6.3: Use of the `blocked` array. The example describes a structure `mutex_t` initialized with the Priority Inheritance protocol. The field `firstblocked` is the first element of the blocked task queue on a specific mutex.

- when a task tries to block a busy mutex, its shadow pointer will be set to the blocking task;
- when a mutex is unlocked, all the task blocked by it are freed, so all the blocked tasks can try to acquire the mutex (the mutex will be locked by the first task blocked scheduled, usually the higher priority task that was blocked);

6.3.2 Private data structures

The PI Module defines the structure `mutex_resource_des` that handle the interface exported by the Resource Modules. The private data structures added by the Module to the interface are the following:

`nlocked` this array stores the number of mutexes that each task currently locks;

`blocked` this array is used to track the blocked tasks on a mutex. Each PI mutex has a pointer to the first blocked task; the other tasks are queued in this structure (look at Figure6.3). The data structure can be allocated locally to the Module because a task can be blocked on only one mutex.

Each mutex handled by the Priority Inheritance protocol uses a dynamically allocated internal data structure called `PI_mutex_t`, that has the following fields:

`owner` When the mutex is free this field is `NIL`, else it is the PID of the task that locks the mutex;

`nblocked` This is the number of tasks actually blocked on a mutex;

`firstblocked` When the field `nblocked` is different from 0 this field is the first task blocked on a mutex (the following tasks can be found following the blocked list).

Finally, to init a PI mutex, a correct parameter must be passed to `mutex_init`. That parameter must be of type `PI_mutexattr_t` and it does not add any other parameter to the default attribute.

6.3.3 Internal and Interface Functions

```
int PI_res_register(RLEVEL l, PID p, RES_MODEL *r);
```

This function always return -1 because it will never be called by the Generic Kernel (because the Module does not accept any Resource Model).

```
void PI_res_detach(RLEVEL l, PID p);
```

This function simply controls that the task that is still ending does not lock any mutexes. If not, an exception is raised. Such a situation is very dangerous because, when a task is died, the shadow data structures are not consistent, and this will probably cause a system crash.

```
int PI_init(RLEVEL l, mutex_t *m, const mutexattr_t *a);
```

This function inits a mutex to be used with the PI Protocol. A structure of type `PI_mutex_t` is allocated and all the fields are initialized..

```
int PI_destroy(RLEVEL l, mutex_t *m);
```

This function should destroy a mutex. The mutex has to be correctly initialized and it must be free (not locked by a task).

```
int PI_lock(RLEVEL l, mutex_t *m);
```

This function should implement a mutex lock. First, a check is done to see if the mutex was initializated statically. In that case, the initialization of the data structures is completed.

At this point, a check is done to see if the task already owns the mutex. If not, a cycle is done. In the body the shadow field is set, and the system is rescheduled. The cycle is needed because when the mutex will be unlocked, all the blocked tasks will be woken up to fight for the locking of the mutex.

Finally, When the mutex will be found free, it is locked.

```
int PI_trylock(RLEVEL l, mutex_t *m);
```

This function is similar to the previous one, except that the task is never blocked if the mutex is busy.

```
int PI_unlock(RLEVEL l, mutex_t *m);
```

This function should free the mutex. First, a check is done to see if the task that unlocks really owns the mutex. Then, the mutex is unlocked and all the blocked tasks are woken up (the shadow field is reset to point to the blocked tasks themselves). Then, the system is rescheduled to see if a preemption should be done (the Module does not know if a preemption will occurs or not, because it does not know which are the modules registered!).

```
void PI_register_module(void);
```

This function will register the Module in the system. This function is very similar to the Scheduling Modules Registration function.

Chapter 7

The Generic Kernel Internals

In this chapter some information are given on the implementation of the Generic Kernel. The objective of this chapter is to give the user enought information about the internal of the kernel to allow an analysis of the source code. The code described in this chapter is contained into the `kernel` directory.

7.1 System Tasks and User Tasks

The Generic Kernel classifies the tasks in the system using two flags of the control field of the task descriptor. The Programming Model of the kernel is a monoprocess multithread model, so each task shares a common memory without any kind of address protection.

The two flags of interest are the `SYSTEM_TASK` and the `NO_KILL` flags:

- If the `SYSTEM_TASK` flag is set a task is a task used internally by the Kernel, otherwise the task is considered an user task;
- If the `NO_KILL` flag is set a task cannot be killed by a `task_kill` or `pthread_cancel` primitive.

These two flags divide the task universe in four sets (look at Figure 7.1 and at Table 7.1):

User Tasks These are the tasks usually created by the user;

Immortal User Tasks These tasks are tasks that the user wants to protect against uncontrolled cancellations. Usually the life of these tasks is not important for the termination of the system; in other words, the system can shut down also if these tasks are not ended;

System Drivers These tasks are handled directly by the Kernel or by some Libraries that implement some important things (for example, the file system controls the hard disks with tasks of this type);

	`SYSTEM_TASK=0`	`SYSTEM_TASK=1`
`NO_KILL=0`	User tasks	System Drivers
`NO_KILL=1`	Immortal User Tasks	System Tasks

Table 7.1: The four sets in whitch the tasks are divided.

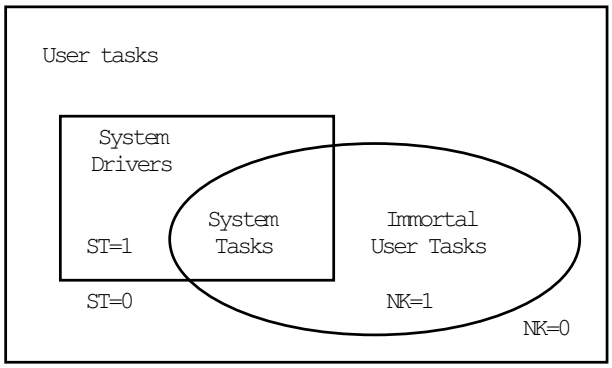

Figure 7.1: The four sets in whitch the tasks are divided; The values of the two flags `SYSTEM_TASK` (ST) and `NO_KILL` (NK) are showed.

System Tasks These are non-critical tasks that have to be always present in the system; the life of the system depends on the life of these tasks. Such a task is for example the dummy task.

System termination can be generated automatically by the Generic Kernel or it can be forced if the user calls the functions `sys_end()` or `sys_abort()`.

The Generic Kernel start the system termination when all User Tasks ends or when all the System Drivers ends.

To do the shutdown in a correct way, the libraries that are implemented using the System Drivers should end in a correct way. Look at Section7.2 for more informations.

7.2 Initialization and Termination

In this section the structure of the function `__kernel_init__` is described in more detail. This function is the function called by the OS Lib at system startup. The interface between the Generic Kernel and the OS Lib is not described here but in section **??**.

7.2.1 Interrupt Disabling

The first thing that is done in the function is the disabling of the interrupts. When the system starts, the OS Lib allocs a context, whose number is stored by the Generic Kernel into the global variable `global_context`. In this startup context the function `__kernel_init__` is called; also the functions that it calls run in that context.

The context used by the tasks will be allocated next, with a call to the OS Lib function `ll_context_create` (this function is called by the Generic Kernel into the primitive `task_create`).

The interrupts will be enabled automatically at the first context change.

7.2.2 Initialization of the Memory Management

After disabling the interrupts, the dynamic memory manager can be initialized. It must be the first thing that is initialized because dynamic memory is used extensively in all the Kernel (and the first place where it is used is the Module Registration).

7.2.3 Initialization of the static data structures

The next step in the in the Kernel startup is the initialization of the staticdata structures. In particular, it will be initialized:

- The task descriptor and the task-specific data;
- The free descriptor queue;
- The arrays that contains the pointers to the Module descriptors;
- The tata structures used to implement POSIX signals;
- The data structures used to call the init functions posted through the function `sys_atrunlevel`.

7.2.4 Resgistration of the Modules in the system

At this point, the system has the interrupts disabled and all static data structures initialized. Now, the Generic kernel needs to know what is the real Module configuration in the system. To handle that, the following function is called:

```
TIME __kernel_register_levels__(void *arg)
```

That function is a user defined function that must call the registration functions of the Scheduling Modules and of the Resource Modules.

It has a parameter that contains a pointer to a `multiboot` structure,that can be used to know some information about the system and about the command line arguments.[1] . These informations can be useful to modify the Module registration dinamically at run time.

The value returned by the function is the system tick that the system will use for the periodic timer initialization. If the value returned is 0 the generic kernel will use the one-shot timer instead (look at Section **??**).

To simplify the developing of the applications, the kernel distribution contains some init examples in the directory `kernel/init`.

In the initialization function only these functions can be used:

- The functions that alloc and free the dynamic memory (described in Section 3.6.3);
- The function `sys_atrunlevel` that can be used to register the initialization and termination functions;
- The functions of the C library exported by the OS Lib;
- The functions that prints some messages on the console, like for example `printk` and `kern_printf`.

The other functions of the Generic Kernel and of the OS Lib can not be used.

For the developers that knows the earlier versions of the Hartik Kernel, the body of the startup function `__kernel_register_levels__` can be thought as the first part of the `main()` function (until the `sys_init()`).

[1] The function `__compute_args__` described in the file `include/kernel/func.h` can be used.

7.2.5 OS Lib initialization

At this point all the data structures are initialized, so the system can go in multitasking mode calling the OS Lib's `ll_init` and `event_init` functions.

7.2.6 Initialization functions call

At this point the system has only one valid context (the `global_context`); there aren't any tasks yet (because nobody could create them before).

To end the initialization part the Generic Kernel calls the initialization functions registered by the Modules registered in the system. Because the OS Lib is initialized, all the function exported by it can be called. Moreover, the primitives `task_create` and `task_activate` can be called.

The initialization functions can be used by the Scheduling Modules to create a startup task set. Tipically the startup task set is composed by two tasks: the `dummy` task (usually created by the `dummy` Scheduling Module [2]) and the task that has the function `__init__` as body (that task is usually created by the Round Robin (`RR`) Scheduling Module). This choice is the tipical situation used in most cases, but is not mandatory. The only thing really important is that there must be a task to schedule after this step.

The function `__init__` is usually contained into the initialization files, just after the `__kernel_register_levels__` function. That function is the body of the first task that is executed in the system. The actions done by that function are tipically two:

- the initialization of some devices and libraries that use some system primitives (for example, the semaphores) and for this reason they must be initialized into a context of a "real" task (and not in the `global_context` where the initialization functions are called). Examples of these libraries are for example the communication ports, and the keyboard and mouse drivers (all these libraries uses the semaphores);
- the call to the `main()` function (in this way the Applications can be writed using the straight C standard)[3].

For the developers that knows the earlier versions of the Hartik Kernel, the body of the `__init__` function can be thought as the part of the `main()` function (from the `sys_init` to the `sys_end()`).

Per quanti fossero familiari con le versioni precedenti del Kernel, il corpo della funzione `__init__` corrisponde più o meno alla parte della funzione `main()` compresa tra la `sys_init` (esclusa) e la `sys_end` finale (esclusa).

7.2.7 First context change

Now, the data structures are initialized, and the first tasks are created. At this point the Generic Kernel simply schedule the first task and dispatch it.

When the first task is scheduled the global context is also saved. Because the global context is not a context of a task, the system will never schedule again the global context until all the user task are finished.

The system will change the context to the global context also when the `sys_end` or the `sys_abort` functions are called to shut down the system. Note that the function `ll_abort` does not change the context to the global context, but it simply change to a safe stack and shut down the OS Lib, without shutting down the Generic Kernel correctly.

[2] This module simply ends the Scheduling Module levels, in a way that there will be always a task to schedule.
[3] The function `__call_main__` described in the file `include/kernel/func.h` can be used.

7.2.8 The shutdown functions

When the last user task ends, or when the `sys_end`, or `sys_abort` function are called, the current context changes to the global context.

At this point the system has to do some operations to shut down the system in a correct way[4].

The operations that have to be called depends on the registered Modules, so the Generic Kernel allows to set, using the function `sys_atrunlevel`, a set of functions to be called at this time.

Usually these functions activates some recovery tasks that will shut down correctly the system. These function should be small, because just after the calls the system will be scheduled again to allow the libraries to shut down using the newly activated threads.

If the shutdown functions are too long and uses a lot of computation time, thete can be some undesirable effects that can put the system in an instable state[5].

7.2.9 Termination request for all user tasks

To speed-up the system termination, the system tries to kill all the user tasks. Because the cancellation is usually deferred (as told by the POSIX standard), this should not cause the instantaneous dead of all tasks when the system returns in multitasking mode.

7.2.10 Second context change

To shut down correctly the system the scheduler must be called again. For the secon time the system exits from the global context. The system usually evolve as follows:

- The user tasks should die (slowly...);
- The shutdown function should give some information to the system tasks so they can finish their work and end.

The system will return to the global context when all system tasks will end or the `sys_abort` will be called. The `sys_end` function does not have any effect in this phase.

7.2.11 Exit functions called before OS Lib termination

When all the tasks end or the `sys_abort` is called the execution returns to the global context. At this point the functions registered through the `sys_atrunlevel` function with the parameter `RUNLEVEL_BEFORE_EXIT` are called.

The mission of these functions is to terminate the cleaning of the system (for example, it may be useful to set the display in text mode if the application uses the graphic modes).

7.2.12 Termination of the OS Lib

At this point the system can end its work. For this reason the function `ll_end` is called; this function frees all the data structures allocated with the `ll_init`. After this call only the function specified in the Section 7.2.4 can be called.

[4]For example, if the File System is used maybe there will be some data that have to be written on the disks

[5]A typical situation is that when the system is rescheduled a task that used a resource miss a deadline; then the Scheduling Module disable its schedulability, and the operation on the device cannot end, and with it all the system!

7.2.13 Exit functions called after OS Lib termination

Finally, the Kernel calls the last functions registered with the `sys_atrunlevel`, that tipically prints some nice messages or simply reboot the computer.

7.3 Task creation and on-line guarantee

The Generic Kernel primitive that creates and guarantees a new task is called `task_createn`. The prototype of that function is the following (the code of the primitive is contained in the file `kernel/create.c`):

```
PID task_createn(char *name, TASK (*body)(), TASK_MODEL *m, ...);
```

The parameter passed with that function are the following:

`name` Symbolic name for the task, used for statistical pourposes;

`body` Pointer to the first instruction of the task;

`m` Pointer to a Task Model for the new task to be created

`...` List of Resource Model pointers terminated with a `NULL` pointer.

The primitive returns the descriptor number associated to the newly created task, or `NIL` if the task cannot be created in the system. In the latter case the variable `errno` is set to a value that explain the typology of the error[6].

There is also a redefinition of the primitive called `task_create` that accept only one Resource Module instead of "...". This redefinition may be useful because usually only a few tasks need more than one Resource Model.

The step followed to create and guarantee correctly a new task are described in the following paragraphs.

The first thing to do is to find a unused task descriptor. Tipically the free descriptors are queued in the `freedesc` queue. During the selection the tasks that are into the freedesc queue but that waits a synchronization with a `task_join` primitive are discarded (look at Section 7.9).

At this point the descriptor chosen is removed from the freedesc queue and initialized with some default values.

Then, a Scheduling Module that can handle the Task Model passed as parameter have to be found. The research is done starting from level 0 and calling the `public_create` function. When a correct Module is found the task is created into that module.

The next step in task creation is the handling of the Resource Models passed. This initialization is done calling the `res_register` function on the Resource Modules registered in the system.

At this point all system components are informed of the Quality of Service required by the new task, and the on-line guarantee can start. The guarantee algorithm cannot be called before registering the Resource Models because in general "hybrid" Modules can be developed (for example, a Module can register itself as Scheduling Module and Resource Module, using the two descriptors...).

Finally, if the task can be guaranteed, the stack memory for the task is allocated (only if needed), the task context is created using the OS Lib function `ll_context_create`), the creation event is registered for the tracer and the task is counted into the user or system task counter.

[6]The error codes are listed in the file `include/bits/errno.h`.

If one of these steps fail, the system will be put in the state preceding the call of the primitive (the functions `public_detach` and `res_detach` are also called).

Looking at the on-line system guarantee, the generic kernel supports a distributed guarantee on all the Scheduling Modules based on the utilization factor paradigm. The system will call the `public_guarantee` function starting from level 0, and passing each time the free bandwidth left by the upper levels. This algorithm is implemented in the `guarantee()` function stored in the file `kernel/kern.c`. That function returns -1 if the task set cannot be guaranteed, 0 otherwise.

This approach allows the implementation in a simple way the on-line guarantee of many algorithms. However, this approach is not suitable to implement more complex algorithms, like for example the Deferrable Server guarantee, the TB* [3] guarantee and others. in these cases two strategies can be used:

- All the system tasks are guarantee off-line, so the guarantee procedure can be disabled at run-time.
- All the algorithms that need a guarantee are developed in a single Sheduling Module, placed at level 0. In this way it can control all the system bandwidth, and a guarantee can be done because the Module knows all the data needed. However, in this way all advantages of the Modularity is lost.

7.4 Task activation

The Generic Kernel, unlike the POSIX standard, decouple the task creation and guarantee and the task activation. This is done because in literature many proofs are given for tasks that are activated at the start of the major cycle. Also, the guarantee function can be heavy and long, unlike the activation that is typically shorter.

The primitives provided by the generic kernel to activate a task are two:

`task_activate` Activation of a single task[7];

`group_activate` Activation of a group of tasks in an atomic way[8].

The Generic kernel provides a mechanism that allow to freeze task activations. That mechanism is inserted in the generic kernel to allow the modular implementation of some shared resource protocols like SRP or similar.

This mechanism use the task descripror control field flag `FREEZE_ACTIVATION`, that stores the freeze state of the activations, and use the task descripror field `frozen_activations`, that stores the number of freed activations for the Generic Kernel.

These primitives are also defined:

`task_block_activation` blocks explicit task activations and activates its counting. The function usually returns 0, or -1 if the task index is not correct.

`task_unblock_activation` enables explicit task activations. it returns -1 if the task had the `FREEZE_ACTIVATION` field disabled, or the number of freezed activations. If there were freezed activations, the primitive does not the activations.

The prototypes presenyted in this Section are showed in Figura 7.2 and they are stored in the files file `kernel/activate.c` and `kernel/blkact.c`.

[7]This primitive can be called also into an OS Lib event and into the global_context (in other words, in the function posted with the primitive `sys_atrunlevel`).

[8]The system is rescheduled only one time, so it can speed-up the activation of a lot of tasks.

```
int task_activate(PID p);
int group_activate(WORD g);
int task_block_activation(PID p);
int task_unblock_activation(PID p);
```

Figure 7.2: Prototypes of the actiovation functions.

7.5 The Scheduler

The steps that the Generic Kernel does when the system is rescheduled are three:

- If when the system is rescheduled a task is running, the end of the slice must be called for that task[9];
- Then, a new task to run must be found (scheduling);
- Finally, the chosen task must be run (dispatching).

These steps are implemented into the system primitives and in the `scheduler()` function stored in the file `kernel/kern.c`. In the following section the three points are showed in detail.

7.5.1 Current slice end for the running task

To specify which are the actions to do at the end of a slice of the running task, the reason af the slice end must be known. Depending on the behaviour of the end of the slice, different actions should be made.

For this reason the Generic Kernel provides different functions that terminates a slice. The cases in that a slice must be ended are the following (into parenthesis the related Task Calls are listed):

1. A new task becomes active in the system, so the Generic Kernel wants to check if a preemption (`public_epilogue`) must be done. This situation can happen in a lot of situations, like for example:
   - a new task is activated with a `task_activate` or `group_activate` primitive;
   - a resource or a mutex is freed, so a task blocked on it is unblocked;
   - a periodic task is reactivated at the beginning of its period;
   - a System Driver is activated because an intrettupt is arrived;
2. The running task finishes its available capacity (`public_epilogue`);
3. The running task blocks itself on a synchronization primitive (`public_block`);
4. The running task ends its instance and it suspend itself with a `task_endcycle` primitive (`public_message`);
5. The running task ends or it is killed by a `task_kill` or `pthread_cancel` primitive (`public_end`);

[9]A task slice is the time interval in that starts when the running task is dispatched and end when the system is scheduled again.

In general the funcions sequence that have to be called is the following:

1. The current time is read into the global variable `schedule_time`[10];
2. The length of the current terminated slice is computed using the variables `schedule_time` and `cap_lasttime`;
3. The computation time of the current slice is accounted to the task (look at Section 7.6);
4. The capacity event (if one is pending) is erased;
5. The Scheduling Module function that handles the termination of the slice for the task is called..

To simplify the writing of the primitives the following approach is implemented: because the preemption rescheduling is the most common situation, the sequence given before that terminates with a call to `public_epilogue` is included as prologue in the `scheduler()` function. That prologue is not executed if the variables `exec` and `exec_shadow` have a `NIL` (`-1`) value when the function is called.

7.5.2 Scheduling

When the previous slice is terminated a new task to schedule must be chosen. The generic scheduling algorithm starts from the Scheduling Module at level 0, calling the function `public_scheduler`, and going through the levels when a Module does not have any task to schedule. The Generic Kernel assumes that there is always a task to schedule[11].

When a task to schedule is found, the function `public_eligible` is called to verify if the task chosen by the scheduler is correct. If the task is not correct, the generic scheduling algorithm restarts from the level that gave the wrong task before (look at Section **??**).

7.5.3 Dispatching

To find the task that should be executed really another step has to be done: the shadow chain of the scheduled task must be followed (look at Section **??**). When the tail of that chain is found, the function `public_dispatch` is called on that task.

Finally, if the task has the `CONTROL_CAP` bit of the task descriptor control field set, a capacity event is posted.

7.6 Execution Time statistics

The Generic Kernel supports the accounting of the task execution times. This is useful because the behaviour of many algorithm proposed in the literature depends widely on the accuracy with that the task capacities are managed.

To enable the Generic Kernel to account the execution time of a task, the user should use the provided macros for the Task Models (look at Section 2.2.1). These macros modifies the `JET_ENABLE` flag into the `control` field of the task descriptor.

[10] That variable is used as temporal reference for the scheduling time. Note that the Generic kernel does not separe the CPU time passed executing user code and system code; all the CPU time is assigned to the user task.

[11] To have always a task to schedule a Scheduling Module called `dummy` is provided that always guarantees the existence of a task to schedule.

```
int jet_getstat(PID p, TIME *sum, TIME *max,
      int *n, TIME *curr);
int jet_delstat(PID p);
int jet_gettable(PID p, TIME *table, int n);
void jet_update_slice(TIME t);
void jet_update_endcycle();
```

Figure 7.3: Primitives for execution time handling and correlated functions used internally by the Generic Kernel.

The Generic Kernel can store some data about a task. In particular, the mean and the maximum execution time of a task and the time consumed by the current instance and of the last `JET_TABLE_DIM` instances.

The prototypes of the Generic Kernel functions are described in Figure7.3. In the following paragraphs these functions are described:

`jet_getstat` This primitive returns some statistical informations; in particular, the informations are stored into the following parameters:

`sum` is the task total execution time since it was created or since the last call to the `jet_delstat` function;

`max` is the maximum time used by a task instance since it was created or since the last call to the `jet_delstat` function;

`n` is the number of terminated instances which sum and max refers to;

`curr` is the total execution time of the current instance.

if a parameter is passed as NULL the information is not returned. The function returns 0 if the PID passed is correct, `-1` if the PID passed does not correspond to a valid PID or the task does not have the `JET_ENABLE` bit set.

`jet_delstat` The primitive voids the actual task execution time data mantained by the Generic Kernel. The function returns 0 if the PID passed is correct, `-1` if the PID passed does not correspond to a valid PID or the task does not have the `JET_ENABLE` bit set.

`jet_gettable` The primitive returns the last n execution times of the task passed as parameter. If the parameter n is less than 0, it returns only the last values stored since the last call to `jet_gettable`. If the value is greater than 0, the function returns the last `min(n, JET_TABLE_DIM)` values registered. The return value is `-1` if the task passed as parameter does not exist or the task does not have the `JET_ENABLE` bit set, otherwise the number of values stored into the array is returned. The table passed as parameter should store at least `JET_TABLE_DIM` elements.

The function used into the Generic Kernel implementation are the following:

`jet_update_slice` updates the current slice of the running task (pointed by the `exec_shadow` field) by t microseconds;

`jet_update_endcycle` updates the execution time of the last instance. When this function is called the last instance is just terminated.

7.7 Cancellation

The POSIX standard provides some mechanisms to enable and disable the cancellation, and to set the cancellation as deferred or asynchronous.

For more informations about the cancellation functions look to the POSIX standard. In Table ?? there are some primitives that are very similar to these of the standard.

The biggest problems in implementing task cancellation into the Generic Kernel are the following:

- The kernel does not have a private stack, and works simply disabling the interrupts into the contexts of the tasks in the system;
- The cancellation functions for a tasks should be called into the stack of the task, so it is not possible to kill another task immediately without changing context;
- The Generic Kernel should abstract from the cancellation points present in the system, because in general it is not possible to handle all the internal structures introduced by a particular cancellation point.

The solution to these problems is proposed in the following Sections.

7.7.1 The task_makefree function

When a task die the flow control of a task is switched to the `task_makefree` function. This function have to call all the cancellation points function, and the key destructors.

The function can be called into the cancellation points (the `task_testcancel` function is called, look at the file `kernel/cancel.c`), and at task termination (look at the `task_create_stub` in the file `kernel/create.c`), and each time a task is scheduled (to test asynchronous cancellation).

The function does the following steps:

- It checks if someone is waiting for the task termination (with a `task_join` primitive);
- It verifies if the task that is terminating is actually using a resource handled with the shadow mechanism (if so an exception is raised);
- It calls some cleanup functions;
- It calls the thread specific data destructors;
- It frees the context[12]and the allocated memory for the stack;
- It calls the `public_end` on the Scheduling Module that owns the task, and the `res_detach` function on the Resource Modules registered in the system;
- It verifies if the end of the task should cause the whole system termination (look at Section 7.1).

[12]Note that the freed context is the running context. this is not a problem because the `task_makefree` is executed with the interrupts disabled, and nobody can use the free memory areas freed.

7.7.2 Cancellation point registration

The last problem to solve is the independence of the Generic Kernel from the Cancellation Points. The objective of the cancellation point registration is to write the code for a cancellation point without modify the primitives that effectively kill a task. The implementations can be depicted with these points:

- The blocking of a task on a cancellation point is implemented through the `public_block` function;
- The task state of a blocked task on a cancellation point is modified to a value visible by the Generic Kernel (usually these names starts with the prefix WAIT_);
- The functions that implements the cancellation points register themselves at their first execution calling the `register_cancellation_point` primitive (this function is defined in the file `kernel/kill.c`). The primitive accepts a function pointer that returns 1 if the task passed as parameter is blocked on the cancellation point handled by the function.
- First, the function that should kill a task sets the `KILL_REQUEST` flag of the control field of the task descriptor; then, it calls the registered cancellation point functions to check if a task is blocked on a cancellation point. If so, the registered function reactivates the blocked task calling the `public_unblock` function.
- The architecture of a cancellation point should guarantee that when a task is woken up a check is made to see if a task is killed. If so, the function internally calls the primitive `task_testcancel` to kill the task.

7.7.3 Cleanups and Thread Specific Data

The POSIX standard provides two primitives, `pthread_cleanup_push` and `pthread_cleanup_pop`, that allows to specify functions to be executed in the case a task has been killed during a section of code delimited by these two functions.

The implementation of these two functions has been done through a macro similar to that contained into the rationale of the POSIX standard.

Their implementation is contained into the files `include/pthread.h` and `include/kernel/func.h`.

The Generic kernel provides also the support for the Thread Specific Data of the POSIX Standard. The implementation of these primitives is not complex and can be found in the file `kernel/keys.c`.

7.8 Signals

The Generic kernel provides a POSIX signal implementation derived from the Flux OSKit [5].

Two aspects need to be described:

- the implementation of the signal interruptable functions:

  To implement these function a registration call is provided in a way similar to the cancellation points. Each time a signal is generated, a check is done to see if some task is blocked on a signal interruptable function. The registration function is called `register_interruptable_point` and it is contained into the file `kernel/signal.c`;

- the correct delivery of the signals:

  a function called `kern_deliver_pending_signals` (defined in the file `kernel/signal.c`) is provided; this function is called into the macro that changes context (the macro `kern_context_load`, defined into the file `include/kernel/func.h`). That function is usually called after a context change, so when a task is rescheduled the pending signals for that task are delivered. Note that in the current version if a task is preempted by a task activated in an interrupt, when the task is rescheduled there will not be any signal dispatching. This IS a bug, and it will be fixed in the next releases of the OS Lib.

Moreover, the OS Kit signal implementation is slightly modified to handle the POSIX message queues and the POSIX realtime timers.

7.9 Task Join

The POSIX standard specifies that a thread return value can be read, if the task is *joinable*, through a call to the primitive `pthread_join` or `task_join`.

In this section the implementation of the primitive `task_join` is described, with all the modification that the implementation has done on the Generic Kernel.

First, the information about the task type (joinable or detached) is stored into the flag `TASK_JOINABLE` of the `control` field of the task descriptor.

Usually the POSIX threads starts in a joinable state and then they can be detached. The Generic kernel follow this line when implementing the `pthread_create`, but with a difference: the default attribute for the task models is detached[13].

The `task_join` primitive implements the POSIX primitive `pthread_join`. It is a cancellation point and it register itself in the Generic kernel the first time it executes.

The main problem in the implementation of this primitive is that a task descriptor correctly terminated can be reused until a join is executed on it. The problem is that in this way the Scheduling Modules should know the internal implementation of the primitive, and this fact may complicate the writing of a Scheduling Module if special task guarantees are implemented.

The implementation tries to avoid these problems in the following way:

- The Scheduling Modules prescind from the task tipology (joinable or detached) and simply inserts a task that terminates in the free queue when the descriptor is no longer needed;
- The `task_makefree` checks if the task is joinable, and if it is the flag `WAIT_FOR_JOIN` in the control field of the task descriptor is set. In any case the context and the stack for the dead process are released;
- A call to `task_create` that tries to alloc a task descriptor that waits for a join and whose descriptor is inserted in the freedesc queue simply discards it, setting the bit `DESCRIPTOR_DISCARDED`, in the `control` field of the task descriptor.
- A call to `task_join` on a task that is already terminated, inserted in the freedesc queue and discarded by the primitive `task_create`, inserts the descriptor in the `freedesc` queue.

This way allow the Scheduling Modules to abstract and remain independent from the implementation of the join primitive.

[13] Note that this does not impact on the standard POSIX implementation, since the task_create is a non-standard function.

7.10 Pause and Nanosleep

The Generic Kernel supports a set of primitives to implement a task suspension. The differences between them are the following:

`sleep` This primitive suspend the execution task for a number of seconds. The task can be woken up by a signal delivery;

`pause` This function suspends the task until a signal is delivered to it;

`nanosleep` This function suspends the running task for a minimum time passed as parameter. The task can be woken up by the dispatch of a signal, in that case the residual time is returned.

7.11 Mutex and condition variables

The Generic kernel provides a set of functions that are similar in the interface with the correspondents POSIX functions that handles mutexes and condition variables.

The extensions to the interface of the Resource Modules described in the previous chapter are used by these primitives to handle different shared resource access protocols in a general way.

Le estensioni apportate all'interfaccia dei Moduli di Gestione delle Risorse descritte nella sezione precedente vengono utilizzate da tali primitive per gestire i vari protocolli di accesso a risorse condivise in modo trasparente.

In particular, the proposed interfaces are the following (for a better description look at the POSIX standard):

```
int mutex_init(mutex_t *mutex, const mutexattr_t *attr);
```

This primitive can be used to init a task descriptor. The attr parameter should be correctly initialized before the call. It can not be NULL.

```
int mutex_destroy(mutex_t *mutex);
```

This function dealloc a mutex.

```
int mutex_lock(mutex_t *mutex);
```

This function implements a blocking wait.

```
int mutex_trylock(mutex_t *mutex);
```

This function implements a non blocking wait.

```
int mutex_unlock(mutex_t *mutex);
```

This functions unlocks a mutex.

```
int cond_init(cond_t *cond);
```

This function initializes a condition variable.

```
int cond_destroy(cond_t *cond);
```

This function destroys a condition variable.

```
int cond_signal(cond_t *cond);
```

This function signals on a condition variable. Only one task is unblocked.

```
int cond_broadcast(cond_t *cond);
```

This function signals on a condition variables, unblocking all the task blocked on the condition variable.

```
int cond_wait(cond_t *cond, mutex_t *mutex);
int cond_timedwait(cond_t *cond, mutex_t *mutex,
                   const struct timespec *abstime);
```

The task that exec this primitive blocks and the mutex passed as parameter is unlocked to be required when the task restarts. There are two versions of the primitive, and one has a timeout to limit blocking times. These functions are cancellation points. If a cancellation request is generated for a task blocked on a condition variable, the task will end after reaquiring the mutex. This implies that each call have to be protected by cleanup functions that should free the mutex in a correct way.

7.12 Other primitives

In this section a set of other primitives are shortly described. They are implemented in the source files contained into the kernel directory.

```
void task_endcycle(void);
```

This primitive terminates the current instance of a task (look at Section 4.1).

```
void task_abort(void);
```

This primitive ends the task.

```
void group_kill(WORD g);
```

This primitive send a kill request to all the tasks that have the group g.

```
TIME sys_time(struct timespec *t);
```

This primitive can be used into the applications to read the system time. Its behaviour is equal to the `ll_gettime` but it is executed with interrupt disabled.

Bibliography

[1] Luca Abeni. Library for operating system development. Technical report, Scuola Superiore di Studi e Perfezionamento S. Anna, 2000.

[2] Luca Abeni and Gerardo Lamastra. The OSLib project. http://oslib.sssup.it.

[3] G. C. Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. In *Proceedings of the 3rd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages pp. 39–48, Como, Italy, September, 8-12 1997.

[4] Robert Davis and Andy Wellings. Dual Priority Scheduling. In *Proceedings of the IEEE Real Time Systems Symposium*, December 1995.

[5] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for os and language research. In *16 ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[6] T. M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9, 1995.

[7] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[8] Lui Sha, Ragunathan Rajkumar, and john P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), september 1990.

[9] IEEE Computer Society, editor. *International Standard ISO/IEC 9945-1: 1996 (E) - IEEE Std 1003.1, 1996 Edition - Information technology - Portable Operating System Interface (POSIX)*. IEEE, 1996.

[10] IEEE Computer Society, editor. *IEEE Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP)*. IEEE, 1998.